

博文视点

博文视点

Broadview
www.broadview.com.cn

Kubernetes

权威指南

第2版

蒋正 吴治辉 王伟
张秀龙 周健勇

000

从Docker到Kubernetes
实践全接触



下一代分布式架构的王者

Kubernetes: The Definitive Guide



中国工信出版集团



电子工业出版社
Publishing House of Electronics Industry
http://www.eipress.com.cn

Kubernetes
实战

电子工业出版社



总 目 录

[Kubernetes实战](#)

[Kubernetes权威指南：从Docker到Kubernetes实践全接触（第2版）](#)



图书在版编目 (CIP) 数据

Kubernetes实战/吴龙辉著.—北京: 电子工业出版社, 2016.5

ISBN 978-7-121-28372-7

I. ①K... II. ①吴... III. ①Linux操作系统—程序设计 IV. ①TP316.85

中国版本图书馆CIP数据核字 (2016) 第055126号

策划编辑: 张春雨

责任编辑: 刘 舫

印 刷: 北京中新伟业印刷有限公司

装 订: 北京中新伟业印刷有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路173信箱 邮编: 100036

开 本: 787×980 1/16 印张: 17.75 字数: 355千字

版 次: 2016年5月第1版

印 次: 2016年5月第1次印刷

定 价: 69.00元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

目 录

[内容简介](#)

[前言](#)

[第1部分 Kubernetes基础篇](#)

[第1章 Kubernetes介绍](#)

[1.1 为什么会有Kubernetes](#)

[1.2 Kubernetes 是什么](#)

[1.3 Kubernetes的发展历史](#)

[1.4 Kubernetes的核心概念](#)

[第2章 Kubernetes的架构和部署](#)

[2.1 Kubernetes的架构和组件](#)

[2.2 部署Kubernetes](#)

[2.3 安装Kubernetes扩展插件](#)

[第3章 Kubernetes快速入门](#)

[3.1 示例应用Guestbook](#)

[3.2 准备工作](#)

[3.3 运行Redis](#)

[3.4 运行Frontend](#)

[3.5 设置Guestbook外网访问](#)

[3.6 清理Guestbook](#)

[第4章 Pod](#)

[4.1 国际惯例的Hello World](#)

[4.2 Pod的基本操作](#)

[4.3 Pod与容器](#)

[4.4 Pod的网络](#)

[4.5 Pod的重启策略](#)

[4.6 Pod的状态和生命周期](#)

[4.7 自定义检查Pod](#)

[4.8 调度Pod](#)

[4.9 问题定位指南](#)

[第5章 Replication Controller](#)

[5.1 持续运行的Pod](#)

[5.2 Pod模板](#)

[5.3 Replication Controller和Pod的关联](#)

[5.4 弹性伸缩](#)

[5.5 自动伸缩](#)

[5.6 滚动升级](#)

[5.7 Deployment](#)

[5.8 一次性任务的Pod](#)

[第6章 Service](#)

[6.1 Service代理Pod](#)

[6.2 Service的虚拟IP](#)

[6.3 服务代理](#)

[6.4 服务发现](#)

[6.5 发布Service](#)

[第7章 数据卷](#)

[7.1 Kubernetes数据卷](#)

[7.2 本地数据卷](#)

[7.3 网络数据卷](#)

[7.4 Persistent Volume和Persistent Volume Claim](#)

[7.5 信息数据卷](#)

[第8章 访问Kubernetes API](#)

[8.1 API对象与元数据](#)

[8.2 如何访问Kubernetes API](#)

[8.3 使用命令行工具kubectl](#)

[第2部分 Kubernetes高级篇](#)

[第9章 Kubernetes网络](#)

[9.1 Docker网络模型](#)

[9.2 Kubernetes网络模型](#)

[9.3 容器间通信](#)

[9.4 Pod间通信](#)

[9.5 Service到Pod通信](#)

[第10章 Kubernetes安全](#)

[10.1 Kubernetes安全原则](#)

[10.2 Kubernetes API的安全访问](#)

[10.3 Service Account](#)

[10.4 容器安全](#)

[10.5 多租户](#)

[第11章 Kubernetes资源管理](#)

[11.1 Kubernetes资源模型](#)

[11.2资源请求和限制](#)

[11.3 Limit Range](#)

[11.4 Resource Quota](#)

[第12章 管理和运维Kubernetes](#)

[12.1 Daemon Pod](#)

[12.2 Kubernetes的高可用性](#)

[12.3 平台监控](#)

[12.4 平台日志](#)

[12.5垃圾清理](#)

[12.6 Kubernetes的Web界面](#)

[第3部分 Kubernetes生态篇](#)

[第13章 CoreOS](#)

[13.1 CoreOS介绍](#)

[13.2 CoreOS工具链](#)

[13.3 CoreOS实践](#)

[第14章 Etcd](#)

[14.1 Etcd介绍](#)

[14.2 Etcd的结构](#)

[14.3 Etcd实践](#)

[第15章 Mesos](#)

[15.1 Mesos介绍](#)

[15.2 Mesos的架构](#)

[15.3 Marathon和K8SM介绍](#)

[15.4 Mesos实践](#)

[返回总目录](#)

内容简介

Docker的流行激活了一直不温不火的PaaS，随之而来的是各类Micro-PaaS的出现，Kubernetes是其中最具代表性的一员，它是Google多年大规模容器管理技术的开源版本。越来越多的企业被迫面对互联网规模所带来的各类难题，而Kubernetes以其优秀的理念和设计正在逐步形成新的技术标准，对于任何领域的运营总监、架构师和软件工程师来说，都是一个绝佳的突破机会。本书以理论加实战的模式，结合大量案例由浅入深地讲解了Kubernetes的各个方面，包括平台架构、基础核心功能、网络、安全和资源管理以及整个生态系统的组成，旨在帮助读者全面深入地掌握Kubernetes+Docker的底层技术堆栈。

前言

随着互联网技术在各领域的广泛应用，所产生的海量数据催生了大数据的诞生。而对于数据中心的需求激活了云计算井喷式的发展，一时间大数据和云计算成为各个企业争夺的战略高地。

在云计算领域的服务模式中，IaaS和SaaS模式已经趋于成熟，因此PaaS就成了全球各大IT巨头和初创公司的焦点，其中的竞争异常激烈。大量的PaaS平台出现，又很快被淘汰，整个行业发生着巨大的迭代更替。正所谓物竞天择，在这样一个激荡变化的背景下，以Docker为代表的容器技术脱颖而出并极速发热，风头无两，大多数主流云厂商已经宣布提供对Docker及其生态系统的支持。容器技术具备融合DevOps的敏捷特性，给云计算市场特别是PaaS市场带来了新的变革力量，Kubernetes就是新一轮变革中产生的一个代表性产品。

Kubernetes是Google开源的容器集群管理系统，它对于容器运行时、编排、常规服务都抽象设计出了准确完整的API，并以此建立一个开放开源的系统，符合企业化需求，每家企业都可以以此搭建出自动化和标准化的底层平台，以优化研发和运营效率。Kubernetes可以说是Google借助着容器领域的爆发，对于其巨大规模数据中心管理的丰富经验的一次实践，旨在建立新的技术业界标准。

展望未来，我们认为将有更多的企业被迫面对互联网规模所带来的各类难题，Kubernetes和Docker技术可以提供应对这些挑战的解决方案。而随着更多企业的加入，会有更多的人以协作方式构建出更强大

的技术堆栈和更多的创新成果，整个行业将朝着更好的方向持续迈进，对此我们乐观其成。

本书特点

本书采用的是理论加实战的模式，结合大量案例由浅入深讲解Kubernetes的各个方面，包括平台架构、基础核心功能、网络、安全和资源管理，以及整个生态系统的组成。技术信息完全来源于Kubernetes开源社区的文档、代码的提炼和总结。本书涉及的Kubernetes内容与官方最新版本同步，包含最新版本的所有新特性说明，并且因为Kubernetes同Docker深度集成，所以本书也会阐述Docker相关的技术话题。

本书的读者对象

本书适用于希望学习和使用Kubernetes以及正在寻找管理数据中心解决方案的工程师和架构师，同时本书可以作为Docker的高级延伸书籍，用于搭建基于Kubernetes+Docker的PaaS平台，实践DevOps。

本书的组织结构

本书在组织结构上分成三部分：Kubernetes基础篇、Kubernetes高级篇和Kubernetes生态篇。基础篇可帮助读者认识Kubernetes，并理解其架构和核心概念，同时能够部署和使用Kubernetes完成基本功能操作。高级篇将深入讲解Kubernetes的网络、安全和资源管理等话题，帮助读者掌握管理Kubernetes的能力。生态篇则介绍与Kubernetes密切

相关的开源软件，包括CoreOS、Etcd和Mesos，使读者对于Kubernetes生态系统有全面的了解。

第1部分

Kubernetes基础篇

第1章 Kubernetes介绍

第2章 Kubernetes的架构和部署

第3章 Kubernetes快速入门

第4章 Pod

第5章 Replication Controller

第6章 Service

第7章 数据卷

第8章 访问Kubernetes API

第1章

Kubernetes介绍

Kubernetes可以说是云计算PaaS领域的集大成者，它借助了最好的帮助，并且在最适当的时间推出，从而得到了最多的关注。本章首先从整个行业背景介绍入手，介绍Kubernetes诞生的前因，并阐述Kubernetes的优势和发展历程，最后说明Kubernetes中的基本概念，帮助读者对Kubernetes有一个初步但全面的认识。

1.1 为什么会有Kubernetes

1.1.1 云计算大潮

云计算（Cloud Computing）作为一个新兴领域，它是多种技术混合演进的结果，在许多大公司和初创企业的共同推动下，发展极为迅速并且持续火热，带来了新一轮的IT变革。云计算带给企业的创新能力和发展空间是不可想象的，我们所有人都正处于云计算大潮中。

云计算从狭义上讲，指IT基础设施的交付和使用模式，即通过网络以按需、易扩展的方式获取所需资源。广义上则指服务的交付和使用模式，通过网络以按需、易扩展的方式获取所需服务。提供资源的网络被形象地比喻成“云”，其计算能力通常是由分布式的大规模集群

和虚拟化技术提供的。而“云”中的计算资源在用户看来是可以扩展，并且可以随时获取、按需使用的。

云计算彻底改变了人们对计算资源的使用方式，有一个形象的比喻说明了云计算革命性的影响：“云”好比一个发电厂，互联网好比是输电线路，只不过这个发电厂对外提供的是IT服务，这种服务将通过互联网传输到千家万户。云计算实现了计算资源从单台发电机供电模式向电厂集中供电模式的转变。

业界根据云计算提供服务资源的类型将其划分为三大类：基础设施即服务（Infrastructure-as-a-Service, IaaS）、平台即服务（Platform-as-a-Service, PaaS）和软件即服务（Software-as-a-Service, SaaS），如图1-1所示。

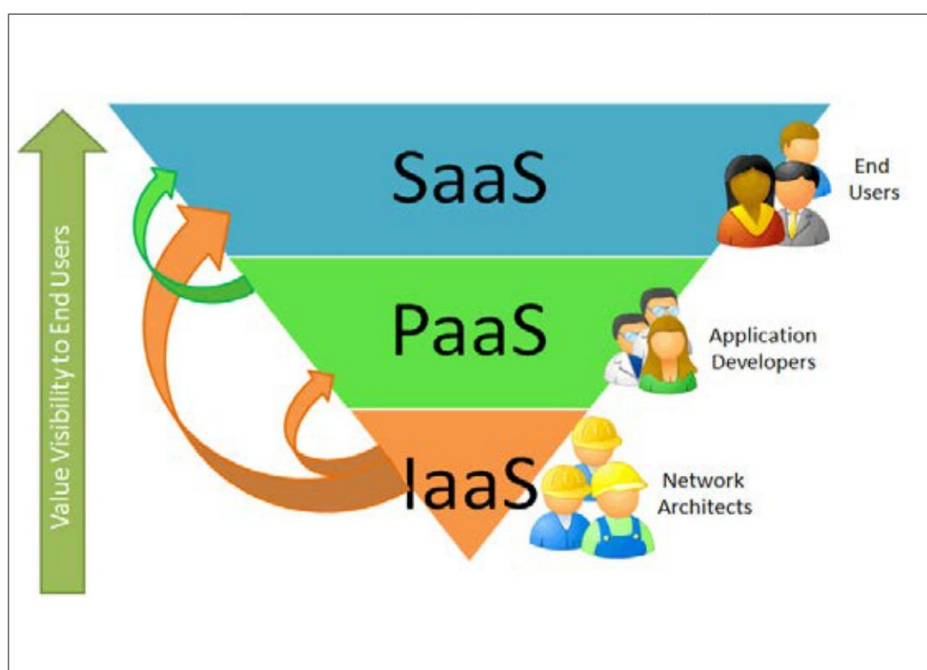


图1-1 云计算的三层架构

• 基础设施即服务

基础设施即服务（IaaS）通过虚拟化和分布式存储等技术，实现了对包括服务器、存储设备、网络设备等各种物理资源的抽象，从而形成了一个可扩展、可按需分配的虚拟资源池。IaaS对外呈现的服务是各种基础设置，例如虚拟机、磁盘以及主机互联而成的网络，这些虚拟机中可以运行Windows系统，也可以运行Linux系统，在用户看来，它与一台真实的物理机是没有区别的。目前最具代表性的IaaS产品有Amazon AWS，其提供了虚拟机EC2和云存储S3等服务。

• 平台即服务

平台即服务（PaaS）为开发者提供了应用的开发环境和运行环境，将开发者从烦琐的IT环境管理中解放出来。自动化应用的部署和运维，使开发者能够集中精力于应用业务开发，极大地提升了应用的开发效率。可以说，PaaS主要面向的是软件专业人员，Google的GAE是PaaS的鼻祖，而Kubernetes可以说是在PaaS的定义范畴内。

• 软件即服务

软件即服务（SaaS）主要面向使用软件的终端用户。一般来说，SaaS将软件功能以特定的接口形式发布，终端用户通过网络浏览器就可以使用软件功能。终端用户将只关注软件业务的使用，除此之外的的工作，如软件的升级和云端实现，对终端用户来说都是透明的。SaaS

是应用最广的云计算模式，比如我们在线使用的邮箱系统和各种管理系统都可以认为是SaaS的范畴。

综上所述，可以简单地概括为：**SaaS**通过网络运行，为最终用户提供应用服务；**PaaS**是一套工具服务，可以为编码和部署应用程序提供快速、高效的服务；**IaaS**包括硬件和软件，例如服务器、存储、网络 and 操作系统。

与SaaS相比，PaaS和IaaS的概念和技术相对较新，图1-2比较了传统IT、IaaS和PaaS。假设现在要上线一项新业务，传统IT的做法就是自下而上地搭建部署、购置硬件、配置网络、安装操作系统、部署中间件系统，到最后业务上线。使用IaaS的客户则无须关心操作系统以下的实现，PaaS更进一步封装操作系统、中间件和运行时，形成标准式的业务发布平台，提供智能化运维能力。这是一种递进式的演化，一步一步地将技术栈分层分级，将资源进行整合管理，可极大提高效率。

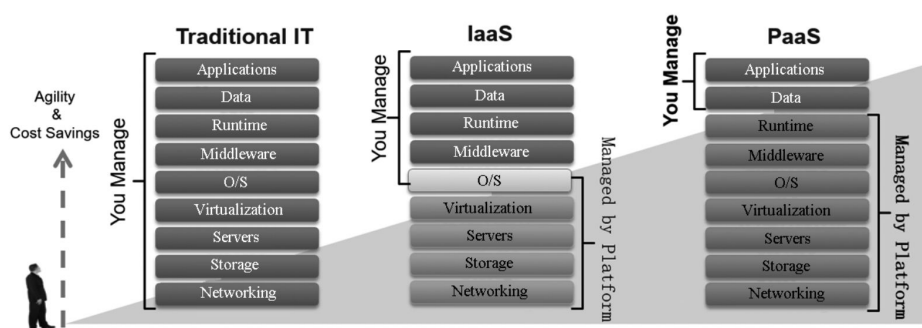


图1-2 传统IT、IaaS和PaaS的比较

正是由于云计算的强大优势，越来越多的公司进入这波潮流中，形成了百家齐放的场面。在云计算的不同层次，在各个行业的不同领域，都涌现出一大批云计算产品，整个云计算市场正在高速发展。

1.1.2 不温不火的PaaS

在SaaS的成熟和IaaS的高速发展催生下，特别是在Amazon、Google、Salesforce、Microsoft等公司的推动下，PaaS得到了长足的发展，越来越多的人开始谈论和关注PaaS，包括运营商、互联网巨头、传统IT厂商、咨询和集成商、IT技术媒体等。但是PaaS的发展可以说是一波三折，可以分为三个阶段。

• 第一代PaaS

比如GAE（Google App Engine）、SAE（Sina App Engine）。这是早期的PaaS，当时并没有PaaS这个概念，现在看来是包含在PaaS范围内的。

• 第二代PaaS

比如Cloud Foundry、Openshift。这是各大IaaS（如Amazon AWS、OpenStack）流行之后，顺势推出的PaaS，并且发展迅速。其中Cloud Foundry是VMware于2011年推出的业界第一个开源PaaS云平台，后来分拆出Pivotal公司进行接管，2014创立Cloud Foundry基金会进行运作。技术和模式相比第一代PaaS都有一定的提高，在云计算大潮中引领了PaaS的发展，一时成为PaaS的代表。华为云、IBM BlueMix、HP Cloud和Dell云服务都采用了Cloud Foundry作为基础。

但是这个阶段的PaaS不管是在市场份额，还是提升速度上都处于弱势，用户对PaaS的兴趣似乎也不大。同时，随着各种云服务之间界限的逐步模糊，一部分人甚至认为PaaS将最终消亡或成为IaaS或者SaaS的一个功能，PaaS处于不温不火的尴尬位置。

• 第三代PaaS

在Docker火爆之后，利用Docker的特性构建出许多PaaS，比如Kubernetes。这些PaaS更加灵活，更加适应企业，逐渐成为PaaS的主力。

1.1.3 Docker的逆袭

Docker是一种Linux容器工具集，它是为构建（Build）、交付（Ship）和运行（Run）分布式应用而设计的。作为DotCloud公司的开源项目，其首发版本的时间是2013年3月。该项目很快就受到欢迎，这也使得DotCloud公司将其品牌改为Docker，并最终将其原有的PaaS业务出售而专注在Docker上，Docker完成了华丽的逆袭。

Docker设计理论来自集装箱，假设交付运行环境如同海运，操作系统如同一艘货轮，每一个在操作系统基础上运行的软件都如同一个集装箱，用户可以通过标准化手段自由组装运行环境，同时集装箱的内容可以由用户自定义，也可以由专业人员制造。这样，交付一个软件，就是一系列标准化组件的集合的交付，如同搭建乐高积木，用户

只需选择合适的积木组合，并且在顶端署上自己的名字，最后这个标准化组件就是用户的应用。

基于这个理念，在技术实现上，**Docker**利用容器（**Container**）来实现类似虚拟机的功能，从而利用更加节省的硬件资源提供给用户更多的计算资源。同虚拟机的方式不同，容器并不是一套硬件虚拟化方法，也无法归属到全虚拟化、部分虚拟化和半虚拟化中的任意一个，而是一个操作系统级虚拟化方法。

Docker容器技术的优势有以下几点。

• 一次构建，到处运行

当将容器固化成镜像后，可以快速地加载到任何环境中部署运行。而构建出来的镜像打包了应用运行所需的程序、依赖和运行环境，这是一个完整可用的应用集装箱，在任何环境下都能保证环境的一致性。

• 容器的快速轻量

容器的启动、停止和销毁都是以秒或毫秒为单位的，并且相比传统的虚拟化技术，使用容器在CPU、内存，网络I/O等资源上的性能损耗都有同样水平甚至更优的表现。

• 完整的生态链

容器技术并不是Docker首创，但是以往的容器实现只关注于如何运行，而Docker站在巨人的肩膀上进行了整合和创新，特别是Docker镜像的设计，完美地为容器从构建、交付到运行提供了完整的生态链支持。

Docker 1.0在2014年6月发布，而且延续了之前每月发布一个版本的节奏。其1.0版本标志着Docker公司认为Docker平台已经足够成熟，并可以被应用到生产环境中。每月的版本更新显示出该项目正在快速发展，比如增加新的特性，解决发现的问题等。

Docker的持续火热是有着坚实的基础来支撑的。Docker吸引了业界众多知名大牌厂家的支持，其中包括Amazon、Canonical、CenturyLink、Google、IBM、Microsoft、New Relic、Pivotal、Red Hat和VMware，这使得只要有Linux的地方，Docker就几乎随处可用。除了这些大厂，许多初创企业也围绕着Docker来发展，或是将他们的发展方向与Docker更好地结合起来。所有这些合作伙伴都驱动着Docker核心项目和周边生态系统的快速发展。

更重要的是Docker的流行和标准化，激活了一直不温不火的PaaS，随之而来的是各类Micro-PaaS的出现，Kubernetes是其中最具代表性的一员。

1.2 Kubernetes 是什么

Kubernetes是Google开源的容器集群管理系统。它构建在Docker技术之上，为容器化的应用提供资源调度、部署运行、服务发现、扩容

扩容等一整套功能，本质上可看作是基于容器技术的Micro-PaaS平台，即第三代PaaS的代表性项目。

Google从2004年起就已经开始使用容器技术了，于2006年发布了Cgroup，而且内部开发了强大的集群资源管理平台Borg和Omega，这些都已经广泛使用在Google的各个基础设施中，而Kubernetes的灵感来源于Google的内部Borg系统，更是吸收了包括Omega在内的容器管理器的经验和教训。

Kubernetes，古希腊语是舵手的意思，也是Cyber的词源，Kubernetes利用Google在容器技术上的实践经验和技術积累，同时吸取Docker社区的最佳实践，已经成为云计算服务的舵手。

Kubernetes有着如下的优秀特性。

• 强大的容器编排能力

Kubernetes可以说是同Docker一起发展起来的，深度集成了Docker，天然适应容器的特点，设计出强大的容器编排能力，比如容器组合、标签选择和服务发现等，可以满足企业级需求。

• 轻量级

Kubernetes遵循微服务架构理论，整个系统划分出各个功能独立的组件，组件之间边界清晰，部署简单，可以轻易地运行在

各种系统和环境中。同时，Kubernetes中的许多功能都实现了插件化，可以非常方便地进行扩展和替换。

• 开放开源

Kubernetes顺应了开放开源的趋势，吸引了大批开发者和公司参与其中，协同工作共同构建生态圈。同时，Kubernetes同OpenStack、Docker等开源社区积极合作、共同发展，企业和个人都可以参与其中并获益。

1.3 Kubernetes的发展历史

Kubernetes自推出后迅速得到关注和参与，2015年7月经过400多位贡献者一年的努力，多达14,000次代码提交，Google正式对外发布了Kubernetes v1.0，意味着这个开源容器编排系统可以正式在生产环境中使用。与此同时，谷歌联合Linux基金会及其他合作伙伴共同成立了CNCF基金会（Cloud Native Computing Foundation），并将Kubernetes作为首个编入CNCF基金会管理体系的开源项目，助力容器技术生态的发展进步。

Kubernetes的发展里程碑如下所示。

- 2014年6月：谷歌宣布Kubernetes开源。
- 2014年7月：Microsoft、Red Hat、IBM、Docker、CoreOS、Mesosphere和Saltstack加入Kubernetes。

- 2014 年 8 月： Mesosphere 宣布将 Kubernetes 作为框架整合到 Mesosphere生态系统中，用于Docker容器集群的调度、部署和管理。

- 2014年8月： VMware加入Kubernetes社区， Google产品经理Craig Mcluckie公开表示， VMware将会帮助Kubernetes实现利用虚拟化来保证物理主机安全的功能模式。

- 2014年11月： HP加入Kubernetes社区。

- 2014年11月： Google容器引擎Alpha启动， 谷歌宣布GCE支持容器及服务， 并以Kubernetes为构架。

- 2015 年 1 月： Google 和 Mirantis 及 伙 伴 将 Kubernetes 引 入 OpenStack， 开发者可以在OpenStack上部署运行Kubernetes应用。

- 2015年4月： Google和CoreOS联合发布Tectonic， 它将Kubernetes和CoreOS软件栈整合在了一起。

- 2015年5月： Intel加入Kubernetes社区， 宣布将合作加速Tectonic软件栈的发展进度。

- 2015年6月： Google容器引擎进入beta版。

- 2015年7月： Google正式加入OpenStack基金会， Kubernetes产品经理Craig McLuckie宣布Google将成为OpenStack基金会的发起人之一， Google将把他的容器计算的专家技术带入OpenStack， 以提高公有云和私有云的互用性。

- 2015年7月： Kubernetes v1.0正式发布。

1.4 Kubernetes的核心概念

1.4.1 Pod

Pod是若干相关容器的组合，Pod包含的容器运行在同一台宿主机上，这些容器使用相同的网络命名空间、IP地址和端口，相互之间能通过localhost来发现和通信。另外，这些容器还可共享一块存储卷空间。在Kubernetes中创建、调度和管理的最小单位是Pod，而不是容器，Pod通过提供更高层次的抽象，提供了更加灵活的部署和管理模式。

1.4.2 Replication Controller

Replication Controller用来控制管理Pod副本（Replica，或者称为实例），Replication Controller确保任何时候Kubernetes集群中有指定数量的Pod副本在运行。如果少于指定数量的Pod副本，Replication Controller会启动新的Pod副本，反之会杀死多余的副本以保证数量不变。另外，Replication Controller是弹性伸缩、滚动升级的实现核心。

1.4.3 Service

Service是真实应用服务的抽象，定义了Pod的逻辑集合和访问这个Pod集合的策略。Service将代理Pod对外表现为一个单一访问接口，

外部不需要了解后端Pod如何运行，这给扩展和维护带来很多好处，提供了一套简化的服务代理和发现机制。

1.4.4 Label

Label是用于区分Pod、Service、Replication Controller的Key/Value对，实际上，Kubernetes中的任意API对象都可以通过Label进行标识。每个API对象可以有多个Label，但是每个Label的Key只能对应一个Value。Label是Service和Replication Controller运行的基础，它们都通过Label来关联Pod，相比于强绑定模型，这是一种非常好的松耦合关系。

1.4.5 Node

Kubernetes属于主从分布式集群架构，Kubernetes Node（简称为Node，早期版本叫作Minion）运行并管理容器。Node作为Kubernetes的操作单元，用来分配给Pod（或者说容器）进行绑定，Pod最终运行在Node上，Node可以认为是Pod的宿主机。

第2章

Kubernetes的架构和部署

Kubernetes遵循微服务架构理论，整个系统划分出各个功能独立的组件，组件之间边界清晰，部署简单，可以轻易地运行在各种系统和环境中。本章将首先介绍Kubernetes的架构和各个组件的功能，然后详细说明如何从头开始部署一套完整的Kubernetes运行环境，包括Kubernetes提供的各种扩展插件（Cluster Add-on）的安装方式。

2.1 Kubernetes的架构和组件

Kubernetes属于主从分布式架构，节点在角色上分为Master和Node，如图2-1所示。

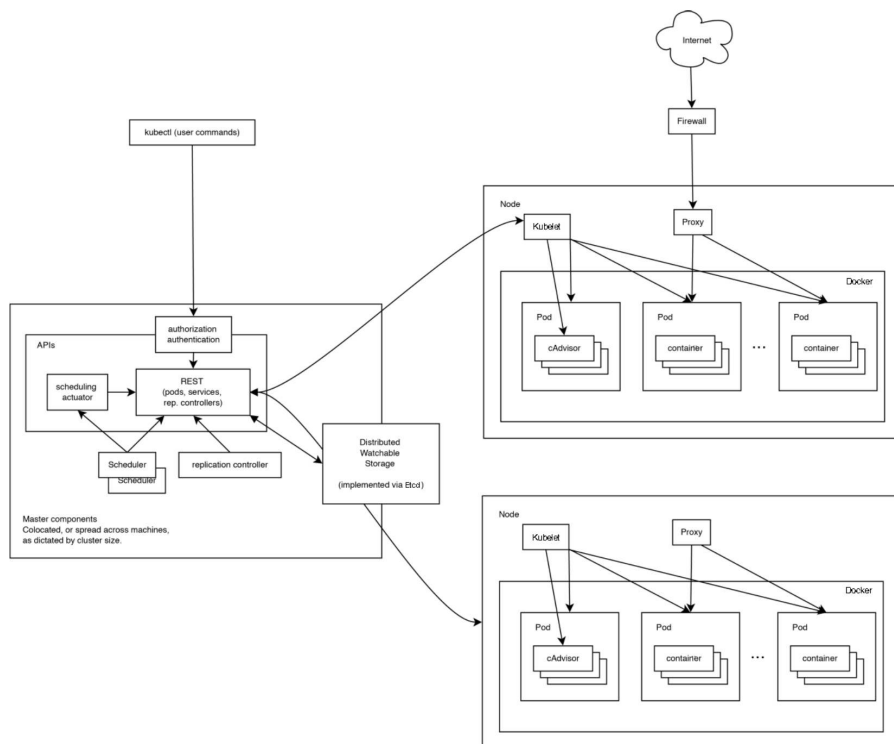


图2-1 Kubernetes的架构

Kubernetes使用Etcd作为存储中间件，Etcd是一个高可用的键值存储系统，灵感来自于ZooKeeper和Doozer，通过Raft一致性算法处理日志复制以保证强一致性。Kubernetes使用Etcd作为系统的配置存储中心，Kubernetes中的重要数据都是持久化在Etcd中的，这使得Kubernetes架构的各个组件属于无状态，可以更简单地实施分布式集群部署。

Kubernetes Master作为控制节点，调度管理整个系统，包含以下组件。

- **Kubernetes API Server:** 作为Kubernetes系统的入口，其封装了核心对象的增删改查操作，以REST API接口方式提供给外部客户和内部组件调用。它维护的REST对象将持久化到Etcd中。

• **Kubernetes Scheduler**: 负责集群的资源调度，为新建的Pod分配机器。这部分工作分出来变成一个组件，意味着可以很方便地替换成其他的调度器。

• **Kubernetes Controller Manager**: 负责执行各种控制器，目前已经实现很多控制器来保证Kubernetes的正常运行，主要包含的控制器如表2-1所示。

表2-1 **Kubernetes 控制器**

控制器	说明
Replication Controller	管理维护Replication Controller，关联Replication Controller和Pod，保证Replication Controller定义的副本数量与实际运行Pod的数量是一致的
Node Controller	管理维护Node，定期检查Node的健康状态，标识出失效的Node
Namespace Controller	管理维护Namespace，定期清理无效的Namespace，包括Namespace下的API对象，像Pod、Service和Secret等
Service Controller	管理维护Service，为LoadBalancer类型的Service创建管理负载均衡器
Endpoints Controller	管理维护Endpoints，关联Service和Pod，创建Endpoints作为Service的后端，当Pod发生变化时，实时刷新Endpoints
Service Account Controller	管理维护Service Account，为每个Namespace创建默认Service Account，同时为Service Account创建Service Account Secret
Persistent Volume Controller	管理维护Persistent Volume和Persistent Volume Claim，为新的Persistent Volume Claim分配Persistent Volume进行绑定，为释放的Persistent Volume执行清理回收
Daemon Set Controller	管理维护Daemon Set，负责创建 Daemon Pod，保证指定的Node上正常运行Daemon Pod
Deployment Controller	管理维护Deployment，关联Deployment和Replication Controller，保证运行指定数目的Pod。当Deployment更新时，控制实现Replication Controller和Pod的更新
Job Controller	管理维护Job，为Job创建一次性任务Pod，保证完成Job指定完成的任务数目
Pod Autoscaler Controller	实现Pod的自动伸缩，定时获取监控数据，进行策略匹配，当满足条件时执行Pod的伸缩动作

Kubernetes Node是运行节点，用于运行管理业务的容器，包含以下组件。

- Kubelet: 负责管控容器，Kubelet会从Kubernetes API Server接收Pod的创建请求，启动和停止容器，监控容器运行状态并汇报给Kubernetes API Server。

- Kubernetes Proxy: 负责为Pod创建代理服务，Kubernetes Proxy会从Kubernetes API Server获取所有的Service，并根据Service信息创建代理服务，实现Service到Pod的请求路由和转发，从而实现Kubernetes层级的虚拟转发网络。

- Docker: Kubernetes Node是容器运行节点，需要运行Docker服务，目前Kubernetes也支持Rocket，这是一款CoreOS开发的类Docker的开源容器引擎，本书只说明Docker。

2.2 部署Kubernetes

Kubernetes能够运行在各个平台上，包括物理机、虚拟机和云平台。在部署方式上，Kubernetes可以在生产环境中大规模地进行分布式部署，也可以简单地运行在单机中以用于测试开发，我们可以根据不同的需求搭建不同的Kubernetes运行环境。

Kubernetes官方和社区针对不同系统和平台提供了自动化部署脚本，具体情况如表2-2所示。

表2-2 Kubernetes 部署支持

底层系统	管理工具	操作系统	网络
GKE			GCE
Vagrant	Vagrant	Fedora	flannel
GCE	GCE	Debian	GCE
Azure	Azure	CoreOS	Weave
Docker Single Node	Docker Single Node	N/A	local
Docker Multi Node	Docker Multi Node	N/A	local
Bare-metal	Bare-metal	Fedora	flannel
Digital Ocean	Digital Ocean	Fedora	Calico
Bare-metal	Bare-metal	Fedora	_none_
Bare-metal	Bare-metal	Fedora	flannel
libvirt	libvirt	Fedora	flannel
KVM	KVM	Fedora	flannel
Mesos/Docker	Mesos/Docker	Ubuntu	Docker
Mesos/GCE	Mesos/GCE		
AWS	AWS	CoreOS	flannel
GCE	GCE	CoreOS	flannel
Vagrant	Vagrant	CoreOS	flannel
Bare-metal (Offline)	Bare-metal (Offline)	CoreOS	flannel
Bare-metal	Bare-metal	CoreOS	Calico
CloudStack	CloudStack	CoreOS	flannel
VMware	VMware	Debian	OVS
Bare-metal	Bare-metal	CentOS	
AWS	AWS	Ubuntu	flannel
OpenStack/HPCloud	OpenStack/HPCloud	Ubuntu	flannel
Joyent	Joyent	Ubuntu	flannel
AWS	AWS	Ubuntu	OVS
Azure	Azure	Ubuntu	OpenVPN
Bare-metal	Bare-metal	Ubuntu	Calico
Bare-metal	Bare-metal	Ubuntu	flannel
Local	Local		
libvirt/KVM	libvirt/KVM	CoreOS	libvirt/KVM
oVirt	oVirt		
Rackspace	Rackspace	CoreOS	flannel

接下来我们将详细说明Kubernetes的部署步骤，同时也会涉及Kubernetes的一些概念，可以进一步帮助读者理解Kubernetes的架构和原理。

2.2.1 环境准备

Kubernetes是一个分布式架构，可灵活地进行安装部署，可以部署在单机，也可以分布式部署。机器可以是物理机，也可以是虚拟机，但是需要运行Linux（x86_64）操作系统，至少1核CPU和1GB内存。

本书准备了4台虚拟机（CentOS 7.0系统）用于部署Kubernetes运行环境，包括一个Etcd、一个Kubernetes Master和三个Kubernetes Node，如表2-3所示。

表2-3 Kubernetes 运行环境

节点	主机名	IP
Etcd	etcd	192.168.3.145
Kubernetes Master	kube-master	192.168.3.146
Kubernetes Node 1	kube-node-1	192.168.3.147
Kubernetes Node 2	kube-node-2	192.168.3.148
Kubernetes Node 3	kube-node-3	192.168.3.149

使用的各种软件版本信息如表2-4所示。

表2-4 软件版本

软件	版本
Kubernetes	1.1.1
Docker	1.7.1
Etcd	2.2.0
Flannel	0.5.1
Open vSwitch	2.3.1

提示

本书后续的实践操作都在此套环境中运行。

2.2.2 运行Etcd

Kubernetes依赖于Etcd，需要先部署Etcd，可以从Github上下载指定版本的Etcd发布包：

```
$ wget https://github.com/coreos/etcd/releases/download/v2.2.0/etcd-v2.2.0-linux-amd64.tar.gz
$ tar xzvf etcd-v2.2.0-linux-amd64.tar.gz
$ cd etcd-v2.2.0-linux-amd64
$ cp etcd /usr/bin/etcd
$ cp etcdctl /usr/bin/etcdctl
```

运行Etcd:

```
$ etcd -name etcd \
-data-dir /var/lib/etcd \
-listen-client-urls http://0.0.0.0:2379, http://0.0.0.0:4001 \
-advertise-client-urls http://0.0.0.0:2379, http://0.0.0.0:4001 \
>> /var/log/etcd.log 2>&1 &
```

Etcd运行后可以查询其健康状态:

```
$ etcdctl -C http://etcd:4001 cluster-health
member ce2a822cea30bfca is healthy: got healthy result from
http://etcd:4001
cluster is healthy
```

2.2.3 获取Kubernetes发布包

Kubernetes发布包可以从Github上下载，本书使用的是Kubernetes V1.1.1版本：

```
$ wget https://github.com/kubernetes/kubernetes/releases/download/v1.1.1/kubernetes.tar.gz
$tar zxvf kubernetes.tar.gz
```

Kubernetes发布包中包含如下内容。

- cluster: Kubernetes自动化部署脚本。
- contrib: Kubernetes非必需程序。
- examples: Kubernetes示例配置文件。
- docs: Kubernetes文档。
- platforms: Kubernetes的命令行工具kubectl。
- server: Kubernetes组件。
- third_party: 第三方程序。

Kubernetes发布包中的server/kubernetes-server-linux-amd64.tar.gz包含各个组件的可执行程序，将这些可执行程序复制到Linux系统目录/use/bin/下：

```
$ cd kubernetes/server
$ tar zxvf kubernetes-server-linux-amd64.tar.gz
$ cd kubernetes/server/bin/
$ find ./ -perm 755 | xargs -i cp {} /usr/bin/
```

2.2.4 运行Kubernetes Master组件

在Kubernetes Master上需要运行以下组件:

- Kubernetes API Server
- Kubernetes Controller Manager
- Kubernetes Scheduler
- Kubernetes Proxy (可选)

Kubernetes API Server

运行Kubernetes API Server:

```
$ kube-apiserver \
--logtostderr=true --v=0 \
--etcd_servers=http://etcd:4001 \
--insecure-bind-address=0.0.0.0 --insecure-port=8080 \
--service-cluster-ip-range=10.254.0.0/16 \
>> /var/log/kube-apiserver.log 2>&1 &
```

Kubernetes Controller Manager

运行Kubernetes Controller Manager:

```
$ kube-controller-manager \  
--logtostderr=true --v=0 \  
--master=http://kube-master:8080 \  
>> /var/log/kube-controller-manager.log 2>&1 &
```

Kubernetes Scheduler

运行Kubernetes Scheduler:

```
$ kube-scheduler \  
--logtostderr=true --v=0 \  
--master=http://kube-master:8080 \  
>> /var/log/kube-scheduler.log 2>&1 &
```

Kubernetes Proxy

运行Kubernetes Proxy:

```
$ kube-proxy \  
--logtostderr=true --v=0 \  
--api_servers=http://kube-master:8080 \  
>> /var/log/kube-proxy.log 2>&1 &
```

2.2.5 运行Kubernetes Node组件

Kubernetes Node上需要运行以下组件:

- Docker
- Kubelet
- Kubernetes Proxy

Docker

Docker 官方社区提供各个平台的安装方法（可参考<http://docs.docker.com/installation/>），很多Linux发行版都陆续对Docker进行了支持，可以使用以下方式快速安装Docker：

```
$ curl -sSL https://get.docker.com/ | sh
```

提示

1. Docker要求Linux 64系统，并且内核至少3.10版本以上。
2. 不同的Kubernetes版本对Docker的版本要求可能会有不同，总体上我们建议使用最新稳定的Docker版本，如果Kubelet发现Docker版本过低，在创建Pod的时候会失败并发出告警日志。

启动Docker：

```
$ docker -d \  
-H unix:///var/run/docker.sock -H 0.0.0.0:2375 \  
>> /var/log/docker.log 2>&1 &
```

Kubelet

运行Kubelet:

```
$ kubelet \
--logtostderr=true --v=0 \
--config=/etc/kubernetes/manifests \
--address=0.0.0.0 \
--api-servers=http://kube-master:8080 \
>> /var/log/kubelet.log 2>&1 &
```

Kubernetes Proxy

运行Kubelet Proxy:

```
$ kube-proxy \
--logtostderr=true --v=0 \
--api_servers=http://kube-master:8080 \
>> /var/log/kube-proxy.log 2>&1 &
```

2.2.6 查询Kubernetes的健康状态

在部署运行各个组件以后，可以通过Kubernetes命令行kubectl查询Kubernetes Master各组件的健康状态:

```
$ kubectl -s http://kube-master:8080 get componentstatus
```

NAME	STATUS	MESSAGE	ERROR
controller-manager	Healthy	ok	nil

scheduler	Healthy	ok	nil
etcd-0	Healthy	{"health": "true"}	nil

以及Kubernetes Node的健康状态:

```
$ kubectl -s http://kube-master:8080 get node
```

NAME		LABELS
STATUS	AGE	
kube-node-1	kubernetes.io/hostname=kube-node-1	Ready
19m		
kube-node-2	kubernetes.io/hostname=kube-node-2	Ready
18m		
kube-node-3	kubernetes.io/hostname=kube-node-3	Ready
18m		

2.2.7 创建Kubernetes覆盖网络

Kubernetes的网络模型要求每一个Pod都拥有一个扁平化共享网络命名空间的IP，称为PodIP，Pod能够直接通过PodIP跨网络与其他物理机和Pod进行通信。要实现Kubernetes的网络模型，需要在Kubernetes集群中创建一个覆盖网络（Overlay Network），联通各个节点，目前可以通过第三方网络插件来创建覆盖网络，比如Flannel和Open vSwitch。

提示

本书中用到的Kubernetes运行环境主要是使用Flannel实现容器覆盖网络的。

Flannel

Flannel是CoreOS团队设计开发的一个覆盖网络工具，可以从Github上下载指定版本的Flannel发布包：

```
$ wget https://github.com/coreos/flannel/releases/download/v0.5.4/flannel-0.5.4-linux-amd64.tar.gz
$ tar xzvf flannel-0.5.4-linux-amd64.tar.gz
$ cd flannel-0.5.4
$ cp flanneld /usr/bin
```

Flannel使用Etcd进行配置，来保证多个Flannel实例之间的配置一致性，所以需要首先在Etcd上进行配置：

```
$ etcdctl -C http://etcd:4001 \
set /coreos.com/network/config '{ "Network": "10.0.0.0/16" }'
```

在Kubernetes的所有节点上运行Flannel：

```
$ flanneld -etcd-endpoints=http://etcd:4001 \
>> /var/log/flanneld.log 2>&1 &
```

Flannel 会为不同 Node 的 Docker 网桥配置不同的 IP 网段以保证 Docker 容器的 IP 在集群内唯一，所以 Flannel 会重新配置 Docker 网桥，需要先删除原先创建的 Docker 网桥：

```
$ iptables -t nat -F
$ ifconfig docker0 down
$ brctl delbr docker0
```

Flannel 运行后会生成一个文件 `subnet.env`，其中包含规划好的 Docker 网桥网段，根据其中的属性重新启动 Docker：

```
$ source /run/flannel/subnet.env
$ docker -d \
-H unix:///var/run/docker.sock -H tcp://0.0.0.0:2375 \
--bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU} \
>> /var/log/docker.log 2>&1 &
```

Open vSwitch

Open vSwitch 是一个高质量的、多层虚拟交换机，使用开源 Apache 2.0 许可协议，由 Nicira Networks 开发。它的目的是让大规模网络自动化可以通过编程扩展，同时仍然支持标准的管理接口和协议。Open vSwitch 是一项非常重要的 SDN 技术，可以实现 Kubernetes 中的覆盖网络。

为保证 Docker 容器的 IP 在集群内唯一，不同 Kubernetes Node 的 Docker 网桥配置成不同的 IP 网段，需要进行规划，如表 2-5 所示。

表 2-5 Kubernetes Node 的 Docker 网桥规划

节点	主机名	IP	Docker网桥
Kubernetes Node 1	kube-node-1	192.168.3.147	10.246.0.1/24
Kubernetes Node 2	kube-node-2	192.168.3.148	10.246.1.1/24
Kubernetes Node 3	kube-node-3	192.168.3.149	10.246.2.1/24

安装运行Open vSwitch服务以后，可以下载一个工具来协助创建Open vSwitch网络：

```
$ wget https://raw.githubusercontent.com/wulonghui/docker-net-tools/master/k8s-ovs-ctl
$ chmod 0750 k8s-ovs-ctl
$ mv k8s-ovs-ctl /usr/bin/
```

在Kubernetes Node上配置~ /k8s-ovs.env:

- Kubernetes Node 1

```
DOCKER_BRIDGE=docker0
CONTAINER_ADDR=10.246.0.1
CONTAINER_NETMASK=255.255.255.0
CONTAINER_SUBNET=10.246.0.0/16

OVS_SWITCH=obr0
```

TUNNEL_BASE=gre

DOCKER_OVS_TUN=tun0

LOCAL_IP=192.168.3.147

NODE_IPS=(192.168.3.147 192.168.3.148 192.168.3.149)

CONTAINER_SUBNETS=(10.246.0.1/24 10.246.1.1/24 10.246.2.1/24)

• Kubernetes Node 2

DOCKER_BRIDGE=docker0

CONTAINER_ADDR=10.246.1.1

CONTAINER_NETMASK=255.255.255.0

CONTAINER_SUBNET=10.246.0.0/16

OVS_SWITCH=obr0

TUNNEL_BASE=gre

DOCKER_OVS_TUN=tun0

LOCAL_IP=192.168.3.148

NODE_IPS=(192.168.3.147 192.168.3.148 192.168.3.149)

CONTAINER_SUBNETS=(10.246.0.1/24 10.246.1.1/24 10.246.2.1/24)

• Kubernetes Node 3

DOCKER_BRIDGE=docker0

CONTAINER_ADDR=10.246.2.1

CONTAINER_NETMASK=255.255.255.0

CONTAINER_SUBNET=10.246.0.0/16

```
OVS_SWITCH=obr0
```

```
TUNNEL_BASE=gre
```

```
DOCKER_OVS_TUN=tun0
```

```
LOCAL_IP=192.168.3.149
```

```
NODE_IPS=(192.168.3.147 192.168.3.148 192.168.3.149)
```

```
CONTAINER_SUBNETS=(10.246.0.1/24 10.246.1.1/24 10.246.2.1/24)
```

配置完成后，先删除原先创建的Docker网桥：

```
$ iptables -t nat -F
```

```
$ ifconfig docker0 down
```

```
$ brctl delbr docker0
```

然后使用k8s-ovs-ctl创建Open vSwitch网络：

```
$ k8s-ovs-ctl setup
```

最后重新运行Docker：

```
$ docker -d \
```

```
-H unix:///var/run/docker.sock -H tcp://0.0.0.0:2375 \
```

```
--bridge=docker0 \
```

```
>> /var/log/docker.log 2>&1 &
```

2.3 安装Kubernetes扩展插件

Kubernetes中提供了许多平台扩展插件（Cluster Add-on），包含在Kubernetes发布包中，可以在Kubernetes上进行安装部署，目前包含以下扩展插件：

- Cluster DNS
- Cluster Monitoring
- Cluster Logging
- Kube UI

提示

Kubernetes平台扩展插件并不是必需的，初次部署可以跳过此章节，待后续阅读中有需要再进行安装。

2.3.1 安装Cluster DNS

Cluster DNS扩展插件用于支持Kubernetes的服务发现机制，Cluster DNS主要包含如下几项。

- SkyDNS：提供DNS解析服务。
- Etcd：用于SkyDNS的存储。

- Kube2sky: 监听Kubernetes，当有新的Service创建时，生成相应记录到SkyDNS。

下载Kubernetes发布包，Cluster DNS扩展插件在Kubernetes发布包的cluster/addons/dns目录下：

```
$ wget https://github.com/kubernetes/kubernetes/releases/download/v1.11.1/kubernetes.tar.gz
$ tar zxvf kubernetes.tar.gz
$ cd kubernetes/cluster/addons/dns
```

通过环境变量配置参数：

```
$ export DNS_SERVER_IP="10.254.10.2"
$ export DNS_DOMAIN="cluster.local"
$ export DNS_REPLICAS=1
```

设置Cluster DNS Server的IP为10.254.10.2，Cluster DNS的本地域为cluster.local，另外需要配置到Kubelet的启动参数中：

```
--cluster-dns=10.254.10.2
--cluster-domain=cluster.local
```

Kubernetes发布包cluster/addons/dns目录下的skydns-rc.yaml.in和skydns-svc.yaml.in是两个模板文件，通过设置的环境变量修改其中相应的属性值，生成Replication Controller和Service的定义文件：

```
$ sed -e "s/{{ pillar\[ 'dns_replicas'\]
}}/${DNS_REPLICAS}/g;s/{{ pillar\[ 'dns_
```

```
domain'\]]}]/${DNS_DOMAIN}/g;" \skydns-rc.yaml.in > skydns-rc.yaml
$ sed -e "s/{{ pillar\[ 'dns_server'\] }}/${DNS_SERVER_IP}/g" skydns-svc.yaml.in > skydns-svc.yaml
```

Cluster DNS Replication Controller的定义文件skydns-rc.yaml如下:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: kube-dns-v9
  namespace: kube-system
  labels:
    k8s-app: kube-dns
    version: v9
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 1
  selector:
    k8s-app: kube-dns
    version: v9
  template:
    metadata:
      labels:
        k8s-app: kube-dns
        version: v9
```

kubernetes.io/cluster-service: "true"

spec:

containers:

- name: etcd

image: gcr.io/google_containers/etcd:2.0.9

resources:

limits:

cpu: 100m

memory: 50Mi

command:

- /usr/local/bin/etcd
- -data-dir
- /var/etcd/data
- -listen-client-urls
- http://127.0.0.1:2379,http://127.0.0.1:4001
- -advertise-client-urls
- http://127.0.0.1:2379,http://127.0.0.1:4001
- -initial-cluster-token
- skydns-etcd

volumeMounts:

- name: etcd-storage

mountPath: /var/etcd/data

- name: kube2sky

image: gcr.io/google_containers/kube2sky:1.11

resources:

limits:

cpu: 100m

```
        memory: 50Mi
args:
# command = "/kube2sky"
- -domain=cluster.local
- -kube_master_url=http://kube-master:8080
- name: skydns
      image: gcr.io/google_containers/skydns:2015-10-13-
8c72f8c
resources:
  limits:
    cpu: 100m
    memory: 50Mi
args:
# command = "/skydns"
- -machines=http://127.0.0.1:4001
- -addr=0.0.0.0:53
- -ns-rotate=false
- -domain=cluster.local.
ports:
- containerPort: 53
  name: dns
  protocol: UDP
- containerPort: 53
  name: dns-tcp
  protocol: TCP
livenessProbe:
  httpGet:
```

```

        path: /healthz
        port: 8080
        scheme: HTTP
        initialDelaySeconds: 30
        timeoutSeconds: 5
    readinessProbe:
        httpGet:
            path: /healthz
            port: 8080
            scheme: HTTP
            initialDelaySeconds: 1
            timeoutSeconds: 5
-   name: healthz
    image: gcr.io/google_containers/exechealthz:1.0
    resources:
        limits:
            cpu: 10m
            memory: 20Mi
    args:
        - -cmd=nslookup kubernetes.default.svc.cluster.local
localhost >/dev/null
        - -port=8080
    ports:
        - containerPort: 8080
          protocol: TCP
    volumes:
        - name: etcd-storage

```

```
emptyDir: {}  
dnsPolicy: Default # Don't use cluster DNS.
```

提示

kube2sky 需要 Service Account 来调用 Kubernetes API，如果 Kubernetes 没有开启 Service Account，可以显式指定 Kubernetes API 的 URL，在 skydns-rc.yaml 中设置 Kube2sky 的启动参数如下所示。

```
args:  
# command = "/kube2sky"  
- -domain=cluster.local  
- -
```

通过定义文件创建 Cluster DNS Replication Controller:

```
$ kubectl create -f skydns-rc.yaml  
replicationcontroller "kube-dns-v9" created
```

Cluster DNS Replication Controller 创建运行 Cluster DNS Pod:

```
$ kubectl get pod --selector k8s-app=kube-dns --namespace=kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
kube-dns-v9-qdck6	4/4	Running	0	2m

Cluster DNS Service 的定义文件 skydns-svc.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: kube-dns
  namespace: kube-system
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "KubeDNS"
spec:
  selector:
    k8s-app: kube-dns
  clusterIP: 10.254.10.2
  ports:
    - name: dns
      port: 53
      protocol: UDP
    - name: dns-tcp
      port: 53
      protocol: TCP
```

通过定义文件创建Cluster DNS Service:

```
$ kubectl create -f skydns-svc.yaml
service "kube-dns" created
```

然后可以查询Cluster DNS Service:


```
$ kubectl get service -l k8s-app=kube-dns --namespace=kube-system
```

NAME	CLUSTER_IP	EXTERNAL_IP	PORT(S)
SELECTOR	AGE		
kube-dns	10.254.10.2	<none>	53/UDP, 53/TCP
app=kube-dns	47s		k8s-

Cluster DNS部署完成后，可以创建Pod进行验证：

```
$ kubectl exec my-pod -- nslookup  
kubernetes.default.cluster.local
```

```
Server: 10.254.10.2
```

```
Address 1: 10.254.10.2
```

```
Name: kubernetes.default.cluster.local
```

```
Address 1: 10.254.0.1
```

2.3.2 安装Cluster Monitoring

Kubernetes提供了Cluster Monitoring作为平台监控支持，Cluster Monitoring的主体是Heapster，一个容器集群的监控收集工具，将收集Kubernetes运行平台的监控数据，支持导入到其他第三方系统，比如InfluxDB和GCE。

下载Kubernetes发布包，Cluster Monitoring扩展插件在Kubernetes发布包的cluster/ addons/cluster-monitoring目录下：

```
$ wget https://github.com/kubernetes/kubernetes/releases/download/v1.11.1/kubernetes.tar.gz
$ tar zxvf kubernetes.tar.gz
$ cd kubernetes/cluster/addons/cluster-monitoring
```

在Kubernetes发布包的cluster/addons/cluster-monitoring目录中包含以下子目录，每个子目录中存放着不同的定义文件。

- standalone: 部署独立运行的Heapster。
- influxdb: 部署Heapster对接InfluxDB。
- google: 部署Heapster对接GCE。
- googleinfluxdb: 部署Heapster对接GCE和InfluxDB。

我们将部署Heapster对接InfluxDB，InfluxDB是一个开源分布式时序、事件和指标数据库，同时，InfluxDB集成Grafana提供图表展示功能。

部署InfluxDB和Grafana

Influxdb&Grafana Replication Controller 的定义文件influxdb/influxdb-grafana-controller.yaml:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: monitoring-influxdb-grafana-v2
```

```

namespace: kube-system
labels:
  k8s-app: influxGrafana
  version: v2
  kubernetes.io/cluster-service: "true"
spec:
  replicas: 1
  selector:
    k8s-app: influxGrafana
    version: v2
  template:
    metadata:
      labels:
        k8s-app: influxGrafana
        version: v2
        kubernetes.io/cluster-service: "true"
    spec:
      containers:
        - image:
gcr.io/google_containers/heapster_influxdb:v0.4
          name: influxdb
          resources:
            limits:
              cpu: 100m
              memory: 200Mi
          ports:
            - containerPort: 8083

```

```

        hostPort: 8083
    - containerPort: 8086
        hostPort: 8086
    volumeMounts:
    - name: influxdb-persistent-storage
      mountPath: /data

- image:
beta.gcr.io/google_containers/heapster_grafana:v2.1.1
  name: grafana
  env:
  resources:
    limits:
      cpu: 100m
      memory: 100Mi
  env:
    # This variable is required to setup templates in
Grafana.

    - name: INFLUXDB_SERVICE_URL
      value: http://monitoring-influxdb:8086
      # The following env variables are required to
make Grafana accessible via
      # the kubernetes api-server proxy. On production
clusters, we recommend
      # removing these env variables,setup auth for
grafana,and expose the grafana
      # service using a LoadBalancer or a public IP.
      #- name: GF_AUTH_BASIC_ENABLED

```

```

        # value: "false"
        #- name: GF_AUTH_ANONYMOUS_ENABLED
        # value: "true"
        #- name: GF_AUTH_ANONYMOUS_ORG_ROLE
        # value: Admin
        #- name: GF_SERVER_ROOT_URL
            # alue: /api/v1/proxy/namespaces/kube-
system/services/monitoring-grafana/
    volumeMounts:
        - name: grafana-persistent-storage
          mountPath: /var
    volumes:
        - name: influxdb-persistent-storage
          emptyDir: {}
        - name: grafana-persistent-storage
          emptyDir: {}

```

提示

influxdb/influxdb-grafana-controller.yaml中注释掉Grafana容器的环境变量:

```

# The following env variables are required to make Grafana
accessible via
# the kubernetes api-server proxy. On production clusters, we
recommend

```

```
# removing these env variables,setup auth for grafana,and
expose the grafana
# service using a LoadBalancer or a public IP.
#- name: GF_AUTH_BASIC_ENABLED
# value: "false"
#- name: GF_AUTH_ANONYMOUS_ENABLED
# value: "true"
#- name: GF_AUTH_ANONYMOUS_ORG_ROLE
# value: Admin
#- name: GF_SERVER_ROOT_URL
# value: /api/v1/proxy/namespaces/kube-
system/services/monitoring-grafana/
```

通过定义文件创建Influxdb&Grafana Replication Controller:

```
$ kubectl create -f influxdb/influxdb-grafana-controller.yaml
replicationcontroller "monitoring-influxdb-grafana-v2" created
```

Influxdb&Grafana Replication Controller创建运行Influxdb&Grafana Pod:

```
$ kubectl get pod --selector k8s-app=influxGrafana --
namespace=kube-system --output wide
```

NAME	READY	STATUS
monitoring-influxdb-grafana-v2-v5e4y	2/2	Running
10m kube-node-2		0

在Pod的查询信息中，属性NODE显示Pod运行在Node kube-node-2上，因为Pod中为Influxdb容器设置了端口映射规则（8083:8083），于是便可以通过Node kube-node-2访问到Influxdb的Web界面，访问地址是http://kube-node-2:8083，如图2-2所示。

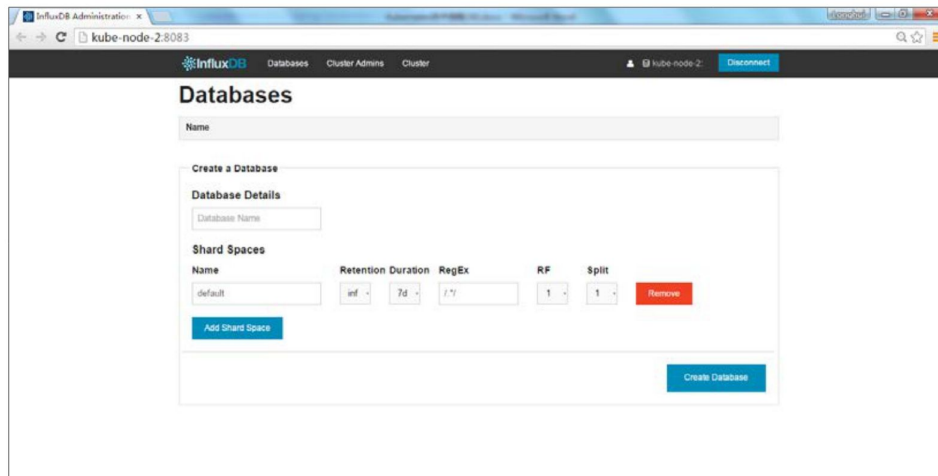


图2-2 访问Influxdb Web界面

Influxdb Service的定义文件influxdb/influxdb-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: monitoring-influxdb
  namespace: kube-system
  labels:
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "InfluxDB"
spec:
  ports:
    - name: http
```

```
    port: 8083
    targetPort: 8083
  - name: api
    port: 8086
    targetPort: 8086
selector:
  k8s-app: influxGrafana
```

通过定义文件创建Influxdb Service:

```
$ kubectl create -f influxdb/influxdb-service.yaml
service "monitoring-influxdb" created
```

Grafana Service的定义文件influxdb/grafana-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: monitoring-grafana
  namespace: kube-system
  labels:
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "Grafana"
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: 3000
```



```
selector:  
  k8s-app: influxGrafana
```

其中设置 Grafana Service 的类型为 NodePort，这样一来，Kubernetes 会为 Service 创建一个端口 NodePort，通过 [Kubernetes Node IP]:[NodePort] 就可以访问到 Service。

通过定义文件创建 Grafana Service:

```
$ kubectl create -f influxdb/grafana-service.yaml
```

```
You have exposed your service on an external port on all nodes  
in your  
cluster. If you want to expose this service to the external  
internet, you may  
need to set up firewall rules for the service port(s)  
(tcp:32075) to serve traffic.
```

```
See          http://releases.k8s.io/release-1.1/docs/user-  
guide/services-firewalls.md for more details.  
service "monitoring-grafana" created
```

从 Grafana Service 创建成功的提示中可以知道 Kubernetes 分配的 NodePort 是 tcp:32075，那么通过 <http://kube-node-2:32075> 可以访问 Grafana，如图2-3所示。

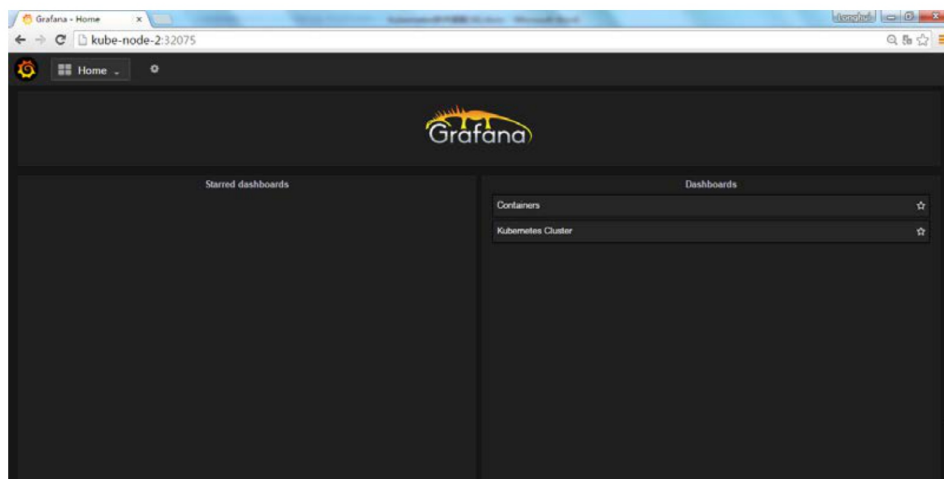


图2-3 访问Grafana

部署Heapster

Heapster Replication Controller 的定义文件 influxdb/heapster-controller.yaml:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: heapster-v10
  namespace: kube-system
  labels:
    k8s-app: heapster
    version: v10
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 1
  selector:
    k8s-app: heapster
```

```
version: v10
template:
  metadata:
    labels:
      k8s-app: heapster
      version: v10
      kubernetes.io/cluster-service: "true"
  spec:
    containers:
      - image: gcr.io/google_containers/heapster:v0.18.2
        name: heapster
        resources:
          limits:
            cpu: 100m
            memory: 300Mi
        command:
          - /heapster
            - --source=kubernetes:http://kube-master:8080?
inClusterConfig=
false&useServiceAccount=false
      - --sink=influxdb:http://monitoring-influxdb:8086
      - --stats_resolution=30s
      - --sink_frequency=1m
```

提示

Heapster 需要 Service Account 来调用 Kubernetes API，如果 Kubernetes 没有开启 Service Account，可以显式指定 Kubernetes API 的 URL，可在 skydns-rc.yaml 中设置 Heapster 的启动参数。

command:

```
- /heapster
      --source=kubernetes:http://kube-master:8080?
inClusterConfig=false&useServiceAccount=
false
  --sink=influxdb:http://monitoring-influxdb:8086
  --stats_resolution=30s
  --sink_frequency=1m
```

通过定义文件创建 Heapster Replication Controller:

```
$ kubectl create -f influxdb/heapster-controller.yaml
replicationcontroller "heapster-v10" created
```

Heapster Replication Controller 创建运行 Heapster Pod:

```
$ kubectl get pod --selector k8s-app=heapster --namespace=kube-
system
```

NAME	READY	STATUS	RESTARTS	AGE
heapster-v10-s8jc1	1/1	Running	0	44s

Heapster 运行后会收集 Kubernetes 的监控数据，导入到 InfluxDB，然后就可以通过 Grafana 查看数据图表展示，如图 2-4 所示。

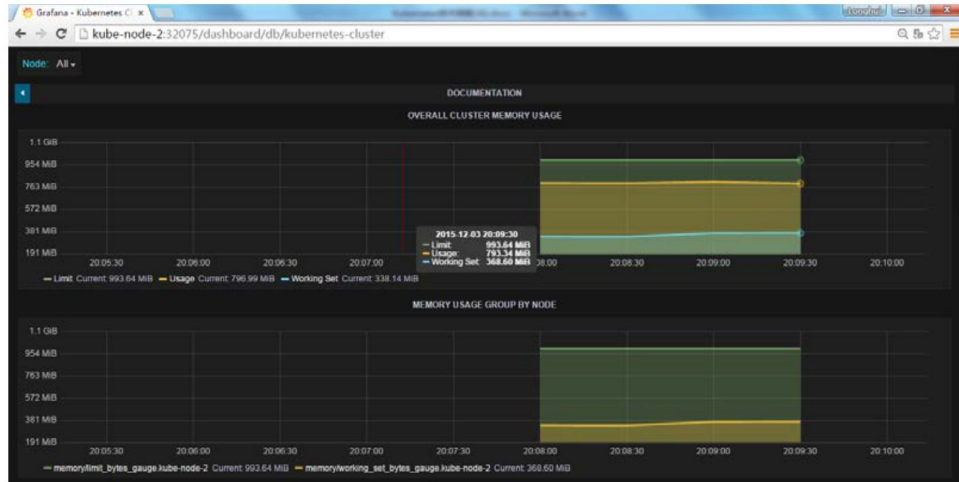


图2-4 Grafana展示Kubernetes监控

Heapster Service的定义文件influxdb/heapster-service.yaml:

```
kind: Service
apiVersion: v1
metadata:
  name: heapster
  namespace: kube-system
  labels:
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "Heapster"
spec:
  ports:
    - port: 80
      targetPort: 8082
  selector:
    k8s-app: heapster
```

通过定义文件创建Heapster Service:

```
$ kubectl create -f influxdb/heapster-service.yaml
service "heapster" created
```

Heapster Service创建成功后就可以通过Kubernetes API Server提供的Proxy接口调用Heapster的Rest API:

```
$ curl http://kube-master:8080/api/v1/proxy/namespaces/kube-
system/services/heapster/api/v1/...
```

2.3.3 安装Cluster Logging

Kubernetes提供了Cluster Logging作为平台日志，Cluster Logging使用Fluentd+Elasticsearch+ Kibana来收集、汇总和展示Kubernetes运行平台的日志。

下载Kubernetes发布包，Cluster Logging扩展插件在发布包的Kubernetes/cluster/addons/ fluentd-elasticsearch目录下:

```
$ wget https://github.com/kubernetes/kubernetes/releases/download/v1.1
.1/kubernetes.tar.gz
$ tar zxvf kubernetes.tar.gz
$ cd kubernetes/cluster/addons/fluentd-elasticsearch
```

部署Elasticsearch

Elasticsearch Replication Controller的定义文件es-controller.yaml:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: elasticsearch-logging-v1
  namespace: kube-system
  labels:
    k8s-app: elasticsearch-logging
    version: v1
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 2
  selector:
    k8s-app: elasticsearch-logging
    version: v1
  template:
    metadata:
      labels:
        k8s-app: elasticsearch-logging
        version: v1
        kubernetes.io/cluster-service: "true"
    spec:
      containers:
        - image: gcr.io/google_containers/elasticsearch:1.7
          name: elasticsearch-logging
          resources:
            limits:
              cpu: 100m
```

```

ports:
  - containerPort: 9200
    name: db
    protocol: TCP
  - containerPort: 9300
    name: transport
    protocol: TCP
volumeMounts:
  - name: es-persistent-storage
    mountPath: /data
volumes:
  - name: es-persistent-storage
    emptyDir: {}

```

通过定义文件创建Elasticsearch Replication Controller:

```
$ kubectl create -f es-controller.yaml
```

```
replicationcontroller "elasticsearch-logging-v1" created
```

Elasticsearch Replication Controller创建运行Elasticsearch Pod:

```
$ kubectl get pod --selector k8s-app=elasticsearch-logging --
namespace=kube-system
```

NAME	READY	STATUS
elasticsearch-logging-v1-joxgq	1/1	Running
48s		0

elasticsearch-logging-v1-ofmn2	1/1	Running	0
48s			

Elasticsearch Service的定义文件es-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch-logging
  namespace: kube-system
  labels:
    k8s-app: elasticsearch-logging
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "Elasticsearch"
spec:
  ports:
    - port: 9200
      protocol: TCP
      targetPort: db
  selector:
    k8s-app: elasticsearch-logging
```

通过定义文件创建Elasticsearch Service:

```
$ kubectl create -f es-service.yaml
service "elasticsearch-logging" created
```

Elasticsearch Service创建成功后，可以通过Kubernetes API Server提供的Proxy接口访问Elasticsearch，如图2-5所示。



图2-5 访问Elasticsearch API

部署Kibana

Kibana Replication Controller的定义文件kibana-controller.yaml:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: kibana-logging-v1
  namespace: kube-system
  labels:
    k8s-app: kibana-logging
    version: v1
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 1
```

```
selector:
  k8s-app: kibana-logging
  version: v1
template:
  metadata:
    labels:
      k8s-app: kibana-logging
      version: v1
      kubernetes.io/cluster-service: "true"
  spec:
    containers:
      - name: kibana-logging
        image: gcr.io/google_containers/kibana:1.3
        resources:
          limits:
            cpu: 100m
        env:
          - name: "ELASTICSEARCH_URL"
            value: "http://elasticsearch-logging:9200"
        ports:
          - containerPort: 5601
            name: ui
            protocol: TCP
```

通过定义文件创建Kibana Replication Controller:

```
$ kubectl create -f kibana-controller.yaml
replicationcontroller "kibana-logging-v1" created
```

Kibana Replication Controller创建运行Kibana Pod:

```
$ kubectl get pod --selector k8s-app=kibana-logging --
namespace=kube-system
```

NAME		READY	STATUS	RESTARTS	AGE
kibana-logging-v1-xqjhz	1/1	Running	1		2m

Kibana Service的定义文件kibana-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: kibana-logging
  namespace: kube-system
  labels:
    k8s-app: kibana-logging
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "Kibana"
spec:
  ports:
    - port: 5601
      protocol: TCP
      targetPort: ui
```

selector:

k8s-app: kibana-logging

通过定义文件创建Kibana Service:

```
$ kubectl create -f kibana-service.yaml
```

```
service "kibana-logging" created
```

Kibana Service创建成功后，就可以通过Kubernetes API Server提供的Proxy接口访问Kibana，如图2-6所示。

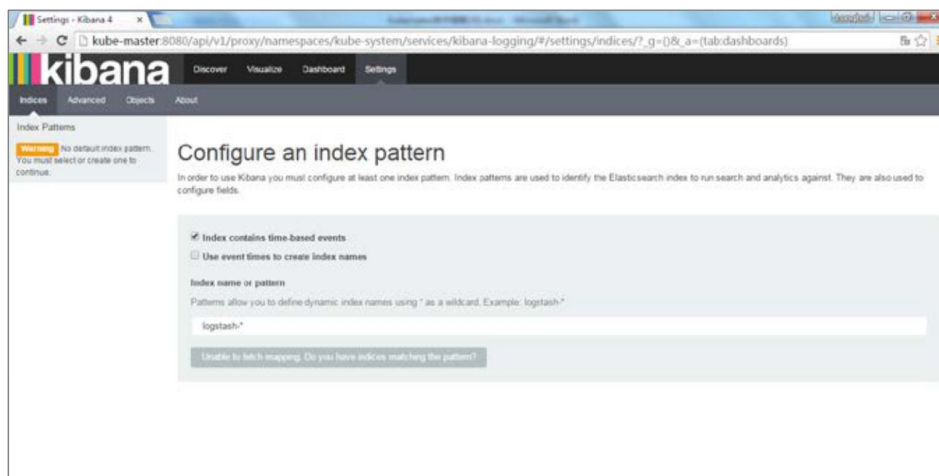


图2-6 访问Kibana

部署Fluentd

Fluentd作为Logging Agent需要运行在所有Kubernetes节点上，我们通过Kubelet将其作为Static Pod（参考12.1.1节）运行。

在所有Kubernetes Node上，在Kubelet的目录/etc/kubernetes/manifests下放入Fluentd Pod的定义文件fluentd-es.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
spec:
  containers:
    - name: fluentd-elasticsearch
      image: gcr.io/google_containers/fluentd-elasticsearch:1.11
      resources:
        limits:
          cpu: 100m
      args:
        - -q
      volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
  volumes:
    - name: varlog
      hostPath:
        path: /var/log
    - name: varlibdockercontainers
```

```
path: /var/lib/docker/containers
```

```
kubectl get pod --selector k8s-app=fluentd-logging --  
namespace=kube-system --output wide
```

NAME		READY	STATUS
RESTARTS	AGE	NODE	
fluentd-elasticsearch-kube-node-1	1/1	Running	0
1m	kube-node-1		
fluentd-elasticsearch-kube-node-2	1/1	Running	0
1m	kube-node-2		
fluentd-elasticsearch-kube-node-3	1/1	Running	0
1m	kube-node-3		

[illegible]

图2-7 Kibana查询日志

2.3.4 安装Kube UI

Kube UI是Kubernetes提供的Web管理界面，下载Kubernetes发布包，Kube UI扩展插件在Kubernetes发布包的cluster/addons/kube-ui目录下：

```
$ wget https://github.com/kubernetes/kubernetes/releases/download/v1.11.1/kubernetes.tar.gz
$ tar zxvf kubernetes.tar.gz
$ cd kubernetes/cluster/addons/kube-ui
```

Kube UI Replication Controller的定义文件kube-ui-rc.yaml：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: kube-ui-v2
  namespace: kube-system
  labels:
    k8s-app: kube-ui
    version: v2
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 1
  selector:
    k8s-app: kube-ui
```



```
version: v2
template:
  metadata:
    labels:
      k8s-app: kube-ui
      version: v2
      kubernetes.io/cluster-service: "true"
  spec:
    containers:
      - name: kube-ui
        image: gcr.io/google_containers/kube-ui:v2
        resources:
          limits:
            cpu: 100m
            memory: 50Mi
        ports:
          - containerPort: 8080
        livenessProbe:
          httpGet:
            path: /
            port: 8080
          initialDelaySeconds: 30
          timeoutSeconds: 5
```

通过定义文件创建Kube UI Replication Controller:

```
$ kubectl create -f kube-ui-rc.yaml
```

```
replicationcontroller "kube-ui-v2" created
```

Kube UI Replication Controller创建运行Kube UI Pod:

```
$ kubectl get pods -l k8s-app=kube-ui --namespace=kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
kube-ui-v2-6g0og	1/1	Running	0	28s

Kube UI Service的定义文件kube-ui-svc.yaml:

```
apiVersion: v1
```

```
kind: Service
```

```
metadata:
```

```
  name: kube-ui
```

```
  namespace: kube-system
```

```
  labels:
```

```
    k8s-app: kube-ui
```

```
    kubernetes.io/cluster-service: "true"
```

```
    kubernetes.io/name: "KubeUI"
```

```
spec:
```

```
  selector:
```

```
    k8s-app: kube-ui
```

```
  ports:
```

```
  - port: 80
```

```
    targetPort: 8080
```

通过定义文件创建Kube UI Service:

```
$ kubectl create -f kube-ui-rc.yaml
```

```
replicationcontroller "kube-ui-v2" created
```

Kube UI Service创建成功后就可以通过Kubernetes API Server的接口访问到Kube UI，如图2-8所示。

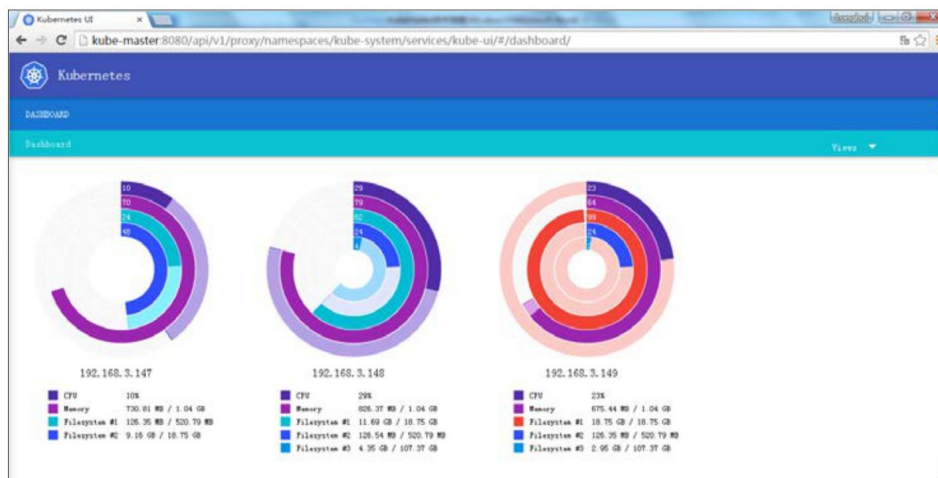


图2-8 访问Kube UI

第3章

Kubernetes快速入门

Kubernetes是容器集群管理系统，为容器化的应用提供资源调度、部署运行、容灾容错和服务发现等功能。本章通过一个完整示例演示Kubernetes的基本功能，其中涉及Pod、Replication Controller和Service这三个Kubernetes基本要素的功能展示，从而帮助读者快速理解Kubernetes。

3.1 示例应用Guestbook

本章要演示的示例应用是一个名叫Guestbook的应用，Guestbook是一个典型的Web应用。Guestbook的部署运行结构如图3-1所示。

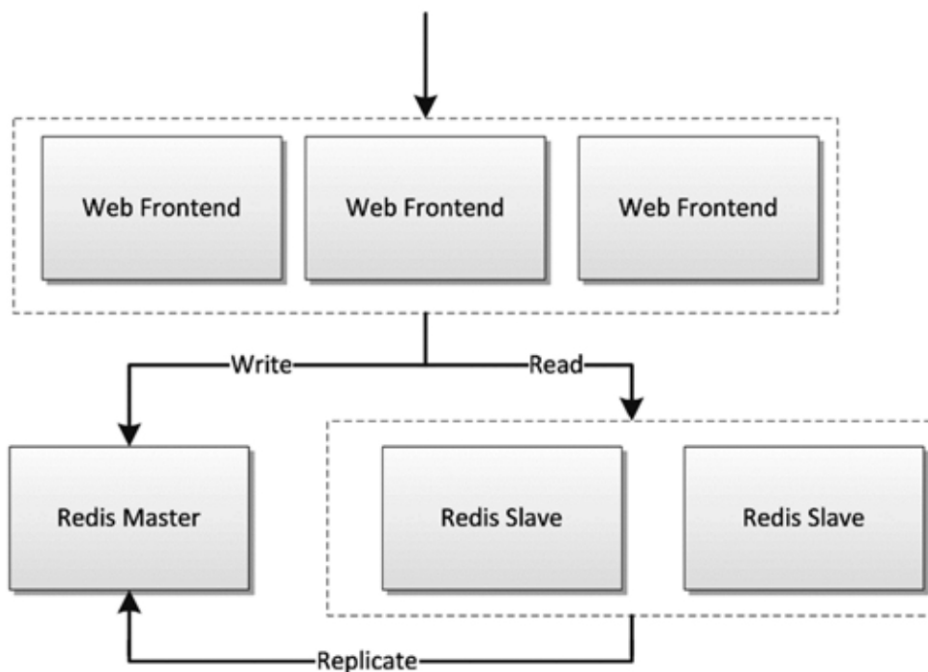


图3-1Guestbook结构

Guestbook包含两部分。

- Frontend

Guestbook的Web前端部分，无状态节点，可以方便伸缩，本例中将运行3个实例。

- Redis

Guestbook的存储部分，Redis采用主备模式，即运行1个Redis Master和2个Redis Slave，Redis Slave会从Redis Master同步数据。

Guestbook提供一个非常简单的功能：在Frontend页面提交数据，Frontend则将数据保存到Redis Master，然后从Redis Slave读取数据显示到页面上。

Guestbook定义文件在Kubernetes发布包的examples/guestbook目录下:

```
$ wget https://github.com/kubernetes/kubernetes/releases/download/v1.1.1/kubernetes.tar.gz
$ tar zxvf kubernetes.tar.gz
$ cd kubernetes/examples/guestbook
```

3.2 准备工作

需要准备一套Kubernetes运行环境, 可参考2.2节的介绍。另外, Kubernetes需要安装Cluster DNS, 可参考2.3.1节的介绍。

本文使用的Kubernetes环境信息如下所示。

```
$ kubectl cluster-info
```

```
Kubernetes master is running at http://k8s-master:8080
KubeDNS is running at http://k8s-master:8080/api/v1/proxy/namespaces/kube-system/services/kube-dns
...
```

```
$ kubectl -s http://kube-master:8080 get componentstatuses
```

NAME	STATUS	MESSAGE	ERROR
controller-manager	Healthy	ok	nil
scheduler	Healthy	ok	nil

```
etcd-0          Healthy   {"health": "true"}   nil
```

```
$ kubectl -s http://kube-master:8080 get nodes
```

NAME		LABELS
STATUS	AGE	
kube-node-1	kubernetes.io/hostname=kube-node-1	Ready
19m		
kube-node-2	kubernetes.io/hostname=kube-node-2	Ready
18m		
kube-node-3	kubernetes.io/hostname=kube-node-3	Ready
18m		

3.3 运行Redis

首先在Kubernetes上部署运行Redis，包括Redis Master和Redis Slave。

3.3.1 创建Redis Master Pod

Pod是Kubernetes的基本处理单元，Pod包含一个或者多个相关的容器，应用将以Pod的形式运行在Kubernetes中（本质上是运行容器）。而Replication Controller能够控制Pod按照指定副本数目持续运行，一般情况下是通过Replication Controller来创建Pod，来保证Pod的可靠性。

Redis Master Replication Controller 的定义文件 redis-master-controller.yaml:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-master
  labels:
    name: redis-master
spec:
  replicas: 1
  selector:
    name: redis-master
  template:
    metadata:
      labels:
        name: redis-master
    spec:
      containers:
        - name: master
          image: redis
          ports:
            - containerPort: 6379
```

在Kubernetes中，主要通过文件来定义API对象，定义文件形式更加清晰，也方便保存和修改，定义文件格式支持JSON和YAML，本书主要使用YAML格式。

定义文件中首先声明了 API 对象的基本属性：API 版本（.apiVersion）、API对象类型（.kind）和元数据（.metadata），定义文件redis-master-controller.yaml中定义了V1版本下一个名称为redis-master的Replication Controller。另外配置了Replication Controller的规格（.spec），其中设置了Pod的副本数（.spec.replicas）和Pod模板（.spec.template）。Pod模板中说明了Pod包含一个容器，该容器使用镜像redis，即运行Redis Master，该Replication Controller将关联1个这样的Pod，而Replication Controller和Pod的关联是通过Label来实现（.spec.selector和.spec.template.metadata.labels）的。

通过定义文件创建Redis Master Replication Controller:

```
$ kubectl create -f redis-master-controller.yaml
replicationcontroller "redis-master" created
```

创建成功后，可查询Redis Master Replication Controller:

```
$ kubectl get replicationcontroller redis-master
```

CONTROLLER		CONTAINER(S)	IMAGE(S)	SELECTOR
REPLICAS	AGE			
redis-master	master	redis	name=redis-master	1
	15s			

Redis Master Replication Controller将会创建1个Redis Master Pod，创建出来的Pod就会带上Label name=redis-master:

```
$ kubectl get pod --selector name=redis-master
```

NAME	READY	STATUS	RESTARTS	AGE
redis-master-vdkfp	1/1	Running	0	31s

Replication Controller在创建出Pod以后，将会保证Pod按照指定副本数目持续运行，而通过Replication Controller也可以对Pod进行一系列操作，包括滚动升级和弹性伸缩等。

3.3.2 创建Redis Master Service

Kubernetes中Pod是变化的，特别是当受到Replication Controller控制的时候，而当Pod发生变化的时候，Pod的IP也是变化的。这就导致了一个问题：在Kubernetes集群中，Pod之间如何互相发现并访问呢？比如我们已经运行了Redis Master Pod，那么Redis Slave Pod如何获取Redis Master Pod的访问地址呢？为此Kubernetes提供了Service来实现服务发现。

Kubernetes中Service是真实应用的抽象，将用来代理Pod，对外提供固定IP作为访问入口，这样通过访问Service便能访问到相应的Pod，而对访问者来说只需知道Service的访问地址，而不需要感知Pod的变化。

上一步中已经运行起Redis Master Pod，现在创建Redis Master Service来代理Redis Master Pod，Redis Master Service的定义文件redis-master-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: redis-master
  labels:
```

```

    name: redis-master
spec:
  ports:
    # the port that this service should serve on
    - port: 6379
      targetPort: 6379
  selector:
    name: redis-master

```

Service 是通过 Label 来关联 Pod 的，在 Service 的定义中，设置.spec.selector为name= redis-master，将关联上Redis Master Pod。

通过定义文件创建Redis Master Service:

```

$ kubectl create -f redis-master-service.yaml
service "redis-master" created

```

创建成功后查看Redis Master Service:

```

$ kubectl get service redis-master

```

NAME	CLUSTER_IP	EXTERNAL_IP	PORT(S)
redis-master	10.254.233.212	<none>	6379/TCP
selector	name=redis-master		
age	13s		

Redis Master Service 的查询信息中显示属性 CLUSTER_IP 为 10.254.233.212，属性 PORT(S) 为 6379/TCP，其中 10.254.233.212 是 Kubernetes 分配给 Redis Master Service 的虚拟 IP，6379/TCP 则是 Service 会转发的端口（通过 Service 定义文件中的.spec.ports[0].port 指定），

Kubernetes会将所有访问10.254.233.212:6379的TCP请求转发到Redis Master Pod中，目标端口是6379/TCP（通过Service定义文件中的spec.ports[0].targetPort指定）。

因为创建了Redis Master Service来代理Redis Master Pod，所以Redis Slave Pod通过Redis Master Service的虚拟IP 10.254.233.212就可以访问到Redis Master Pod，但是如果只是硬配置Service的虚拟IP到Redis Slave Pod中，这样还不是真正的服务发现，Kubernetes提供了两种发现Service的方法。

• 环境变量

当Pod运行的时候，Kubernetes会将之前存在的Service的信息通过环境变量写到Pod中，以Redis Master Service为例，它的信息会被写到Pod中：

```
REDIS_MASTER_SERVICE_HOST=10.254.233.212
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.254.233.212:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.254.233.212:6379
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.254.233.212
```

这种方法要求Pod必须在Service之后启动，之前启动的Pod没有这些环境变量。采用DNS方式就没有这个限制。

• DNS

当有新的Service创建时，就会自动生成一条DNS记录，以Redis Master Service为例，有一条DNS记录：

```
redis-master => 10.254.233.212
```

使用这种方法，Kubernetes需要安装Cluster DNS，可参考2.3.1节的介绍。

3.3.3 创建Redis Slave Pod

通过Replication Controller可创建Redis Slave Pod，将创建两个Redis Slave Pod。Redis Slave Replication Controller的定义文件redis-slave-controller.yaml：

```
kind: ReplicationController
metadata:
  name: redis-slave
  labels:
    name: redis-slave
spec:
  replicas: 2
  selector:
    name: redis-slave
  template:
```

```

metadata:
  labels:
    name: redis-slave
spec:
  containers:
  - name: worker
    image: gcr.io/google_samples/gb-redisslave:v1
    env:
      - name: GET_HOSTS_FROM
        value: dns
        # If your cluster config does not include a dns
service, then to
        # instead access an environment variable to find the
master
        # service's host, comment out the 'value: dns' line
above, and
        # uncomment the line below.
        # value: env
    ports:
      - containerPort: 6379

```

Redis Slave Replication Controller定义中设置Pod副本数为2，而Pod模板中包含一个容器，容器使用镜像gcr.io/google_samples/gb-redisslave:v1，该镜像实际上是基于镜像redis重写了启动脚本，将作为Redis Master的备节点启动，启动脚本如下：

```
if [[ ${GET_HOSTS_FROM:-dns} == "env" ]]; then
    redis-server --slaveof ${REDIS_MASTER_SERVICE_HOST} 6379
else
    redis-server --slaveof redis-master 6379
fi
```

其中通过环境变量 `GET_HOSTS_FROM` 来控制使用环境变量或者 DNS 方式来发现 Redis Master Server。

提示

镜像 `gcr.io/google_samples/gb-redisslave:v1` 的 Dockerfile 参考：

<https://github.com/kubernetes/kubernetes/tree/v1.1.1/examples/guestbook/redis-slave>

通过定义文件创建 Redis Slave Replication Controller:

```
$ kubectl create -f redis-slave-controller.yaml
replicationcontroller "redis-slave" created
```

创建成功后，查询 Redis Slave Replication Controller:

```
$ kubectl get replicationcontroller redis-slave
```

CONTROLLER	CONTAINER(S)	IMAGE(S)
SELECTOR	REPLICAS	AGE

```
redis-slave      worker      gcr.io/google_samples/gb-  
redisslave:v1    name=redis-slave 2      4s
```

Redis Slave Replication Controller创建运行两个Redis Slave Pod:

```
$ kubectl get pod --selector name=redis-slave
```

NAME	READY	STATUS	RESTARTS	AGE
redis-slave-7g617	1/1	Running	0	24s
redis-slave-8dhc2	1/1	Running	0	24s

3.3.4 创建Redis Slave Service

创建 Redis Slave Service 来代理 Redis Slave Pod , Redis Slave Service的定义文件redis- slave-service.yaml:

```
apiVersion: v1  
kind: Service  
metadata:  
  name: redis-slave  
  labels:  
    name: redis-slave  
spec:  
  ports:  
    # the port that this service should serve on  
    - port: 6379  
  selector:  
    name: redis-slave
```


通过定义文件创建Redis Slave Service:

```
$ kubectl create -f redis-slave-service.yaml
service "redis-slave" created
```

查询Redis Slave Service:

```
$ kubectl get service redis-slave
```

NAME	CLUSTER_IP	EXTERNAL_IP	PORT(S)
redis-slave	10.254.108.203	<none>	6379/TCP
selector	age		
name=redis-slave	4s		

3.4 运行Frontend

3.4.1 创建Frontend Pod

通过Frontend Replication Controller来创建Frontend Pod，将创建3个Frontend Pod。

Frontend Replication Controller 的定义文件 frontend-controller.yaml:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
```

```
labels:
  name: frontend
spec:
  replicas: 3
  selector:
    name: frontend
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google_samples/gb-frontend:v3
          env:
            - name: GET_HOSTS_FROM
              value: dns
              # If your cluster config does not include a dns
service, then to
              # instead access environment variables to find
service host
              # info, comment out the 'value: dns' line above, and
uncomment the
              # line below.
              # value: env
      ports:
        - containerPort: 80
```

Frontend Replication Controller的定义中设置Pod副本数为3，Pod模板包含一个容器，容器使用镜像gcr.io/google_samples/gb-frontend:v3，这是一个PHP实现的Web应用，简单地说将写数据到Redis Master，并从Redis Slave中读取数据：

```
<?
```

```
set_include_path('.:usr/local/lib/php');
```

```
error_reporting(E_ALL);
```

```
ini_set('display_errors', 1);
```

```
require 'Predis/Autoloader.php';
```

```
Predis\Autoloader::register();
```

```
if (isset($_GET['cmd']) === true) {
```

```
    $host = 'redis-master';
```

```
    if (getenv('GET_HOSTS_FROM') == 'env') {
```

```
        $host = getenv('REDIS_MASTER_SERVICE_HOST');
```

```
    }
```

```
    header('Content-Type: application/json');
```

```
    if ($_GET['cmd'] == 'set') {
```

```
        $client = new Predis\Client([
```

```
            'scheme' => 'tcp',
```

```
            'host'    => $host,
```

```
            'port'    => 6379,
```

```

]);

$client->set($_GET['key'], $_GET['value']);
print('{ "message": "Updated" }');
} else {
    $host = 'redis-slave';
    if (getenv('GET_HOSTS_FROM') == 'env') {
        $host = getenv('REDIS_SLAVE_SERVICE_HOST');
    }
    $client = new Predis\Client([
        'scheme' => 'tcp',
        'host'    => $host,
        'port'    => 6379,
    ]);

    $value = $client->get($_GET['key']);
    print('{ "data": "' . $value . '" }');
}
} else {
    phpinfo();
} ?>

```

代码中是通过环境变量 `GET_HOSTS_FROM` 来控制使用环境变量方式还是 DNS 方式来发现 Redis Master Server 和 Redis Slave Server 的。

提示

镜像gcr.io/google_samples/gb-frontend:v3的Dockerfile参考:

<https://github.com/kubernetes/kubernetes/tree/v1.1.1/examples/guestbook/php-redis>

通过定义文件创建Frontend Replication Controller:

```
$ kubectl create -f frontend-controller.yaml
```

```
replicationcontroller "frontend" created
```

创建成功后, 查询Frontend Replication Controller:

```
$ kubectl get replicationcontroller frontend
```

CONTROLLER		CONTAINER(S)	IMAGE(S)
SELECTOR	REPLICAS	AGE	
frontend	php-redis	gcr.io/google_samples/gb-frontend:v3	
name=frontend	3	9s	

Frontend Replication Controller创建运行3个Frontend Pod:

```
$ kubectl get pod --selector name=frontend
```

NAME	READY	STATUS	RESTARTS	AGE
frontend-1670s	1/1	Running	0	22s
frontend-2t6sw	1/1	Running	0	21s
frontend-ehbxc	1/1	Running	0	21s

3.4.2 创建Frontend Service

创建Frontend Service代理Frontend Pod，Frontend Service的定义文件frontend-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  ports:
    # the port that this service should serve on
    - port: 80
  selector:
    name: frontend
```

通过定义文件创建Frontend Service:

```
$ kubectl create -f frontend-service.yaml
service "frontend" created
```

查询Frontend Service:

```
$ kubectl get service frontend
```

NAME	CLUSTER_IP	EXTERNAL_IP	PORT(S)
SELECTOR	AGE		
frontend	10.254.69.78	<none>	80/TCP
name=frontend	22s		

3.5 设置Guestbook外网访问

至此Guestbook就已经运行在Kubernetes上，但是还有一件很重要的事情要解决，用户该如何访问Guestbook Frontend呢？是否通过Frontend Service的虚拟IP 10.254.69.78？但是Service的虚拟IP是由Kubernetes虚拟出来的内部网络，而外部网络是无法寻址到的，这时候就需要增加一层网络转发，即外网到内网的转发。实现方式有很多种，我们这里采用一种叫作NodePort的方式来实现。即Kubernetes将会在每个Node上设置端口，称为NodePort，通过NodePort端口可以访问到Pod。

修改Frontend Service的定义文件frontend-service.yaml，设置spec.type为NodePort:

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  type: NodePort
  ports:
    # the port that this service should serve on
    - port: 80
  selector:
    name: frontend
```

重新创建Frontend Service:

```
$ kubectl replace -f frontend-service.yaml --force
```

You have exposed your service on an external port on all nodes in your

cluster. If you want to expose this service to the external internet, you may

need to set up firewall rules for the service port(s) (tcp:31505) to serve traffic.

See <http://releases.k8s.io/release-1.1/docs/user-guide/services-firewalls.md> for more details.

service "frontend" created

Frontend Service 创建成功信息中说明了已经创建 NodePort (tcp:31505), 那么通过任何一个Node的IP和NodePort (tcp:31505)即可访问到Guestbook Frontend。然后在界面的输入框中输入任意字符串并提交, 字符串将会被保存并显示在最下方, 如图3-2所示。

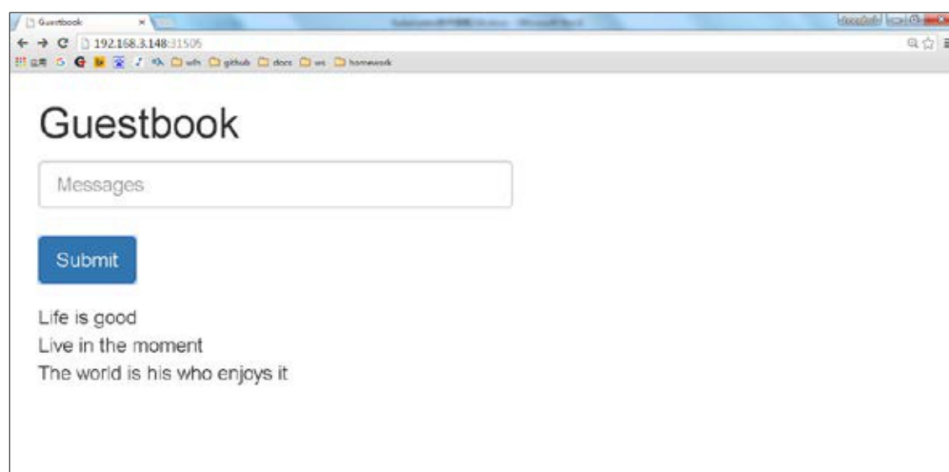


图3-2 Guestbook界面

3.6 清理Guestbook

清理Guestbook，只需要分别删除创建出的Replication Controller和服务Service:

```
$ kubectl delete replicationcontroller redis-master redis-slave  
frontend
```

```
replicationcontroller "redis-master" deleted  
replicationcontroller "redis-slave" deleted  
replicationcontroller "frontend" deleted
```

```
$ kubectl delete service redis-master redis-slave frontend
```

```
service "redis-master" deleted  
service "redis-slave" deleted  
service "frontend" deleted
```

第4章

Pod

Pod是Kubernetes的基本操作单元，也是应用运行的载体。整个Kubernetes系统都是围绕着Pod展开的，比如如何部署运行Pod、如何保证Pod的可靠性、如何访问Pod等。另外，Pod是一个或多个相关容器的集合，这可以说是一大创新点，提供了一种容器组合的模型，当然也使得在Pod的操作和生命周期管理上稍有不同。本章将围绕Pod进行详细讲解，首先介绍如何运行一个最基本的Pod，然后由浅入深地说明Pod的各个方面，包括资源隔离、网络、生命周期管理和调度。

4.1 国际惯例的Hello World

不免俗地我们也将以Hello World作为开始，创建一个简单的Hello World Pod，运行一个输出“Hello World”的容器，Hello World Pod的定义文件hello-world-pod.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  restartPolicy: OnFailure
  containers:
```

```
- name: hello
  image: "ubuntu:14.04"
  command: ["/bin/echo", "Hello", "World"]
```

定义文件中描述了Pod的属性和行为，其中的主要要素如下所示。

- **apiVersion**: 声明Kubernetes的API版本，目前是v1。
- **kind**: 声明API对象的类型，这里类型是Pod。
- **metadata**: 设置Pod的元数据。
 - **name**: 指定Pod的名称，Pod名称必须在Namespace内唯一。
- **spec**: 配置Pod的具体规格。
 - **restartPolicy**: 设置Pod的重启策略。
 - **containers**: 设置Pod中容器的规格，数组形式，每一项定义一个容器。
 - name**: 指定容器的名称，在Pod的定义中唯一。
 - image**: 设置容器镜像。
 - command**: 设置容器的启动命令

Hello World Pod的定义中设置了一个容器，名称是hello，使用镜像ubuntu:14.04，同时启动命令是[/bin/echo,"Hello","World"]，实际上这类似我们使用docker run命令运行容器：

```
$ docker run --name hellop ubuntu:14.04 /bin/echo Hello World
Hello World
```

需要注意的是，因为容器输出完Hello World就会退出，这是一次性执行的，所以在Pod的定义中，`.spec.restartPolicy`设置为`OnFailure`，即在容器正常退出的情况下不会重新创建容器。

通过定义文件创建Hello World Pod:

```
$ kubectl create -f hello-world-pod.yaml
```

```
pod "hello-world" created
```

创建成功后，可以查询Hello World Pod:

```
$ kubectl get pod hello-world
```

NAME	READY	STATUS	RESTARTS	AGE
hello-world	0/1	ExitCode:0	0	1m

然后可以查询Pod输出:

```
$ kubectl logs hello-world
```

```
Hello World
```

最后删除Hello World Pod:

```
$ kubectl delete pod hello-world
```

```
pod "hello-world" deleted
```

4.2 Pod的基本操作

4.2.1 创建Pod

4.1节中我们使用`kubectl create`命令创建了Pod，Kubernetes中大部分API对象都是通过`kubectl create`命令创建的。

如果Pod的定义存在错误，`kubectl create`会打印错误信息，现有一个Pod的错误定义文件`error-pod.yaml`：

```
apiVersion: v1
kind: Pod
metadata:
spec:
  restartPolicy: Maybe
  containers:
  - name: hello
    image: "ubuntu:14.04"
    entrypoint : ["/bin/echo", "Hello", "World"]
```

通过定义文件创建Pod，将会创建失败并提示相应的错误信息：

```
$ kubectl create -f error-pod.yaml
```

```
The Pod "" is invalid.
```

```
* metadata.name: required value, Details: name or generateName
is required
* spec.restartPolicy: unsupported value 'Maybe' , Details:
supported values: Always, OnFailure, Never
```

4.2.2 查询Pod

最常用的查询命令就是**kubectl get**，可以查询一个或者多个Pod的信息，现在查询指定Pod：

```
$ kubectl get pod my-pod
```

NAME	READY	STATUS	RESTARTS	AGE
my-pod	1/1	Running	0	10s

查询显示的字段含义如下所示。

- **NAME**： Pod的名称。
- **READY**： Pod的准备状况，右边的数字表示Pod包含的容器总数目，左边的数字表示准备就绪的容器数目。
- **STATUS**： Pod的状态。
- **RESTARTS**： Pod的重启次数。
- **AGE**： Pod的运行时间。

其中Pod的准备状况指的是Pod是否准备就绪以接收请求，Pod的准备状况取决于容器，即所有容器都准备就绪了，Pod才准备就绪。这时候Kubernetes的代理服务才会添加Pod作为分发后端，而一旦Pod的准备状况变为false（至少一个容器的准备状况变为false），Kubernetes会将Pod从代理服务的分发后端移除，即不会分发请求给该Pod。

默认情况下，**kubectl get**只是显示Pod的简要信息，以下方式可用于获取Pod的完整信息：

```
$ kubectl get pod my-pod --output json #用JSON格式显示Pod的完整信息
```

```
$ kubectl get pod my-pod --output yaml #用YAML方式显示Pod的完整信息
```

另外，`kubectl get`支持以Go Template方式过滤出指定的信息，比如查询Pod的运行状态：

```
$ kubectl get pods my-pod --output=go-template --template={{.status.phase}}  
Running
```

另一个命令`kubectl describe`支持查询Pod的状态和生命周期事件：

```
$ kubectl describe pod my-pod  
Name:      my-pod  
Namespace: default  
Image(s):  nginx  
Node:      kube-node-2/192.168.3.148  
Start Time:   Sun, 22 Nov 2015 18:00:34 +0800  
Labels:      <none>  
Status:      Running  
Reason:  
Message:  
IP:          10.0.62.37  
Replication Controllers: <none>  
Containers:  
  container1:  
    Container ID:
```

```
docker://60387df7e5f2c781ed508084ffa3579f05482c6b465abb3d4fcae0a  
de064b813
```

```
Image: nginx
```

```
Image ID:
```

```
docker://6886fb5a9b8d73b12d842bab8f9a6941c36094c2974abddb685d54c  
9d99e37da
```

```
State: Running
```

```
Started: Sun, 22 Nov 2015 18:00:42 +0800
```

```
Ready: True
```

```
Restart Count: 0
```

```
Environment Variables:
```

```
Conditions:
```

```
Type Status
```

```
Ready True
```

```
Volumes:
```

```
...
```

```
Events:
```

```
...
```

查询显示的字段含义如下所示。

- Name: Pod的名称。
- Namespace: Pod的Namespace。
- Image(s): Pod使用的镜像。
- Node: Pod所在的Node。

- Start Time: Pod的起始时间。
- Labels: Pod的Label。
- Status: Pod的状态。
- Reason: Pod处于当前状态的原因。
- Message: Pod处于当前状态的信息。
- IP: Pod的PodIP。
- Replication Controllers: Pod对应的Replication Controller。
- Containers: Pod中容器的信息。
 - Container ID: 容器的ID。
 - Image: 容器的镜像。
 - Image ID: 镜像的ID。
 - State: 容器的状态。
 - Ready: 容器的准备状况（true表示准备就绪）。
 - Restart Count: 容器的重启次数统计。
 - Environment Variables: 容器的环境变量。
- Conditions: Pod的条件，包含Pod的准备状况（true表示准备就绪）。
- Volumes: Pod的数据卷。
- Events: 与Pod相关的事件列表。

4.2.3 删除Pod

可以通过kubectl delete命令删除Pod:

```
$ kubectl delete pod my-pod
```

另外，kubectl delete命令可以批量删除全部Pod:

```
$ kubectl delete pod --all
```

4.2.4 更新Pod

Pod在创建之后如果希望更新Pod，可以在修改Pod的定义文件后执行:

```
$ kubectl replace /path/to/my-pod.yaml
```

但是因为Pod的很多属性是没办法修改的，比如容器镜像，这时候可以通过kubectl replace命令设置--force参数，等效于重建Pod。

4.3 Pod与容器

在Docker中，容器是最小处理单位，增删改查的对象是容器，容器是一种虚拟化技术，容器之间是隔离的，隔离是基于Linux Namespace实现的，Linux内核中提供了6种Linux Namespace隔离的系统调用，如表4-1所示。

表4-1 Linux Namespace

Linux Namespace	系统调用参数	隔离内容
UTS	CLONE_NEWUTS	主机名与域名
IPC	CLONE_NEWIPC	信号量、消息队列和共享内存
PID	CLONE_NEWPID	进程编号
Network	CLONE_NEWNET	网络设备、网络栈、端口等
Mount	CLONE_NEWNS	挂载点（文件系统）
User	CLONE_NEWUSER	用户和用户组

而在Kubernetes中，Pod包含一个或者多个相关的容器，Pod可以认为是容器的一种延伸扩展，一个Pod也是一个隔离体，而Pod包含的一组容器又是共享的（当前共享的Linux Namespace包括：PID、Network、IPC和UTS）。除此之外，Pod中的容器可以访问共同的数据卷来实现文件系统的共享，所以Kubernetes中的数据卷是Pod级别的，而不是容器级别的。

Pod是容器的集合，容器是真正的执行体。相比原生的容器接口，Pod提供了更高层次的抽象，但是Pod的设计并不是为了运行同一个应用的多个实例，而是运行一个应用多个紧密联系的程序。而每个程序运行在单独的容器中，以Pod的形式组合成一个应用。相比于在单个容器中运行多个程序，这样的设计有以下好处。

- 透明性：将Pod内的容器向基础设施可见，底层系统就能向容器提供如进程管理和资源监控等服务，这样能给用户带来极大便利。
- 解绑软件的依赖：这样单个容器可以独立地重建和重新部署，可以实现独立容器的实时更新。
- 易用性：用户不需要运行自己的进程管理器，也不需负责信号量和退出码的传递等。

- 高效性：因为底层设备负责更多的管理，容器因而能更轻量化。

在Pod中可以详细地配置如何运行一个容器，就像我们使用docker run命令运行容器一样，接下来结合实例说明如何在Pod中定义容器。

4.3.1 镜像

运行容器必须先指定镜像，镜像的名称则遵循Docker的命名规范。运行容器前需要本地存在对应的镜像，如果镜像不存在，会从Docker镜像仓库下载。Kubernetes中可以选择镜像的下载策略，支持的策略如下。

- Always: 每次都下载最新的镜像。
- Never: 只使用本地镜像，从不下载。
- IfNotPresent: 只有当本地没有的时候才下载镜像。

Kubernetes Node是容器运行的宿主机，Pod被分配到Node之后，会根据镜像下载策略选择是否下载镜像。有时候网络下载是一个较大的开销，可以根据需要自行选择策略，但是无论如何要确保镜像在本地或者镜像仓库存在，否则Pod无法运行。

Pod定义中一个容器镜像的配置示例如下所示：

```
name: hello
image: "ubuntu:14.04"
imagePullPolicy: Always
```

Docker镜像仓库也称为Docker注册服务器（Docker Registry），它包括Docker公共镜像仓库（Docker Hub）和私有镜像仓库。

例如对于ubuntu:14.04，它实际上等效于docker.io/ubuntu:14.04，即从Docker公共镜像仓库下载镜像。而对于myregistry.com/ubuntu:14.04来说，myregistry.com是Docker私有镜像仓库地址，即从myregistry.com下载镜像。

使用Docker私有镜像仓库，往往需要进行认证。一种方法是在所有的Node上手工操作docker login [registry]进行登录认证；另一种方法是在Pod中添加Image Pull Secret用于认证，Image Pull Secret是一种kubernetes.io/dockercfg类型的Secret，Kubernetes用来进行Docker镜像仓库的认证，具体操作方法如下所示。

1. 首先登录Docker私有注册服务器,例如myregistry.com:

```
$ docker login myregistry.com
```

```
WARNING: The Auth config file is empty
```

```
Username: admin
```

```
Password:
```

```
Email: wlh6666@qq.com
```

```
WARNING: login credentials saved in /root/.dockercfg.
```

```
Login Succeeded
```

登录成功后，Docker会生成一个文件.dockercfg:

```
$ echo $(cat ~/.dockercfg)
```

```
{  "myregistry.com": {  "auth":  "d2F1YWRtaW46d2F1QDEyMw==" ,  
"email": "wlh6666@qq.com" } }
```

2. 使用Base64编码.dockercfg内容:

```
$ export DOCKER_CFG_DATA=`cat ~/.dockercfg | base64`
```

根据编码后的内容创建Image Pull Secret的定义文件:

```
$ cat > ./image-pull-secret.yaml <<EOF
```

```
apiVersion: v1
```

```
kind: Secret
```

```
metadata:
```

```
  name: myregistrykey
```

```
data:
```

```
  .dockercfg: ${DOCKER_CFG_DATA}
```

```
type: kubernetes.io/dockercfg
```

```
EOF
```

通过定义文件创建Image Pull Secret:

```
$ kubectl create -f ./image-pull-secret.yaml
```

```
secret "myregistrykey" created
```

然后在Pod的定义中通过.spec.imagePullSecrets 添加 Image Pull Secret:

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  name: redis
```

```
spec:
```

```
containers:
  - name: redis
    image: myregistrykey.com/redis
restartPolicy: Always
imagePullSecrets:
  - name: myregistrykey
```

提示

系统管理员可以设置全局的 Image Pull Secret，创建 Pod 的时候会自动添加上 Image Pull Secret。这样一来就不需要每个 Pod 显示配置，可参考10.3.3节。

4.3.2 启动命令

启动命令用来说明容器是如何运行的，在Pod的定义中可以设置容器启动命令和参数，4.1节中的Hello World Pod就对容器设置了启动命令：

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  restartPolicy: Never
```

```
containers:
```

```
- name: hello
```

```
  image: "ubuntu:14.04"
```

```
  command: ["/bin/echo", "Hello", "World"]
```

另外，容器的启动命令也可以配置为：

```
command: ["/bin/echo"]
```

```
args: ["Hello", "World"]
```

在使用`docker run`命令运行容器的时候，如果没有指定容器的启动命令，容器则使用Docker镜像默认的启动命令。这一般是通过Dockerfile中的CMD和ENTRYPOINT进行设置的，这是非常容易混淆的两个概念，假设镜像image1的Dockerfile声明CMD命令如下：

```
FROM ubuntu
```

```
CMD ["echo"]
```

运行容器：

```
$ docker run image1 echo hello
```

```
hello
```

另一个镜像image2的Dockerfile声明ENTRYPOINT命令如下：

```
FROM ubuntu
```

```
ENTRYPOINT ["echo"]
```

运行容器：


```
$ docker run image2 echo hello
echo hello
```

从示例中可以看出，**CMD**命令是可覆盖的，**docker run**指定的启动命令会把**CMD**设置的命令覆盖。而**ENTRYPOINT**设置的命令只是一个入口，**docker run**指定的启动命令作为参数传递给**ENTRYPOINT**设置的命令，而不是进行替换。

在Pod的定义中，**command**和**args**都是可选项，将和Docker镜像的**ENTRYPOINT**和**CMD**相互作用，生成最终容器的启动命令，具体规则如下所示：

- 如果容器没有指定 **command** 和 **args**，则容器使用镜像的 **ENTRYPOINT**和**CMD**作为启动命令运行。
- 如果容器指定了 **command**，而没有指定**args**，则容器忽略镜像的 **ENTRYPOINT**和**CMD**，使用指定的**command**作为启动命令运行。
- 如果容器没有指定**command**，只是指定了**args**，容器将使用镜像的**ENTRYPOINT**和**CMD**作为启动命令运行。
- 如果容器指定了**command**和**args**，则容器使用**command**和**args**作为启动命令运行。

表4-2枚举了4条规则的相应示例。

表4-2 容器命令示例

镜像ENTRYPOINT	镜像CMD	容器command	容器args	最终启动命令
[/ep-1]	[foo bar]	NULL	NULL	[ep-1 foo bar]
[/ep-1]	[foo bar]	[/ep-2]	NULL	[ep-2]
[/ep-1]	[foo bar]	<not set>	[zoo boo]	[ep-1 zoo boo]
[/ep-1]	[foo bar]	<not set>	[zoo boo]	[ep-2 zoo boo]

4.3.3 环境变量

Pod定义中可以设置容器运行时的环境变量：

env:

- name: PARAMETER_1
value: value_1
- name: PARAMETER_2
value: value_2

对于Hello World Pod，可以通过设置环境变量的方式进行改造：

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  restartPolicy: Never
  containers:
  - name: hello
    image: "ubuntu:14.04"
    env:
    - name: MESSAGE
```

```
    value: "hello world"
  command: ["/bin/sh", "-c"]
  args: ["/bin/echo \"${MESSAGE}\""]
```

在一些场景下，Pod中的容器希望获取本身的信息，比如Pod的名称、Pod所在的Namespace等。在Kubernetes中提供了Downward API获取这些信息，并且可以通过环境变量告诉容器目前支持的信息。

- Pod的名称: `metadata.name`。
- Pod的Namespace: `metadata.namespace`。
- Pod的PodIP: `status.podIP`。

现在创建一个Pod并通过环境变量来获取Downward API，Pod的定义文件`downwardapi-env.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: downwardapi-env
spec:
  containers:
    - name: test-container
      image: ubuntu:14.04
      command: ["/bin/bash", "-c", "while true; do sleep 5; done"]
      env:
        - name: MY_POD_NAME
          valueFrom:
```

```

        fieldRef:
          fieldPath: metadata.name
-   name: MY_POD_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
-   name: MY_POD_IP
    valueFrom:
      fieldRef:
        fieldPath: status.podIP

```

Pod创建运行后，查询Pod的输出，过滤出配置的3个环境变量：

```
$ kubectl exec downwardapi-env env|grep MY_POD
```

```
MY_POD_NAME=downwardapi-env
```

```
MY_POD_NAMESPACE=default
```

```
MY_POD_IP=10.0.10.103
```

4.3.4 端口

在使用docker run运行容器的时候往往通过--publish/-p参数设置端口映射规则，同样的，可以在Pod的定义中设置容器的端口映射规则，比如下面这个Pod的设置容器nginx的端口映射规则为0.0.0.0:80->80/TCP：

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
  name: my-nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - name: web
      containerPort: 80
      protocol: TCP
      hostIP: 0.0.0.0
      hostPort: 80
```

在Pod的定义中，通过`.spec.containers[].ports[]`设置容器的端口，数组形式，每一项的参数如下所示。

- **name**: 设置端口名称，必须在Pod内唯一，当只配置一个端口的时候，这是一个可选项，当配置多个端口的时候，这是一个必选项。
- **containerPort**: 必选项，设置在容器内的端口，有效值范围为0~65536（不包括0和65536）。
- **protocol**: 可选项，设置端口的协议，TCP或者UDP，默认是TCP。
- **hostIP**: 可选项，设置在宿主机上的IP，默认绑定到所有可用的IP接口上，即0.0.0.0。

- **hostPort**: 可选项，设置在宿主机上的端口，如果设置则进行端口映射，有效值范围为0~ 65536（不包括0和65536）。

使用宿主机端口需要考虑端口冲突问题，幸运的是，Kubernetes在调度Pod的时候，会检查宿主机端口是否冲突。比如两个Pod都需要使用宿主机端口80，那么调度的时候就会将这两个Pod调度到不同Node上。不过，如果所有Node的端口都被占用了，那么Pod调度会失败。

4.3.5 数据持久化和共享

容器是临时存在的，如果容器被销毁，容器中的数据将会丢失。为了能够持久化数据以及共享容器间的数据，Docker提出了数据卷（Volume）的概念。简单来说，数据卷就是目录或者文件，它可以绕过默认的联合文件系统，而以正常的文件或者目录的形式存在于宿主机上。

在使用docker run运行容器的时候，我们经常使用参数--volume/-v创建数据卷，即将宿主机上的目录或者文件挂载到容器中。即使容器被销毁，数据卷中的数据仍然保存在宿主机上。

一方面，在Kubernetes中对Docker数据卷进行了扩展，支持对接第三方存储系统。另一方面，Kubernetes中的数据卷是Pod级别的，Pod中的容器可以访问共同的数据卷，实现容器间的数据共享。

我们再次对Hello World Pod进行改造。在Pod中声明创建数据卷，Pod中的两个容器将共享数据卷，容器write写入数据，容器hello读出数据，Hello World Pod的定义文件hello-world-pod.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  restartPolicy: Never
  containers:
    - name: write
      image: "ubuntu:14.04"
      command: ["bash", "-c", "echo \"Hello World\" >> /data/hello"]
      volumeMounts:
        - name: data
          mountPath: /data
    - name: hello
      image: "ubuntu:14.04"
      command: ["bash", "-c", "sleep 10; cat /data/hello"]
      volumeMounts:
        - name: data
          mountPath: /data
  volumes:
    - name: data
      hostPath:
        path: /tmp
```

可以看到在Pod定义中，`.spec.volumes`配置了一个名称为data的数据卷，数据卷的类型是`hostPath`，使用宿主机的目录`/tmp`。Pod中的两个容器都通过`.spec.containers[]. volumeMounts`来设置挂载数据卷到容器中

的路径/data。容器write将往/data/hello写入“Hello World”，容器hello等待一会儿，然后读取文件/data/hello的数据显示，即输出“Hello World”。这样一来就实现了两个容器的数据共享。Kubernetes数据卷提供了非常丰富的持久化支持，详情可参考7.3节。

4.4 Pod的网络

Pod中的所有容器网络都是共享的，一个Pod中的所有容器中的网络是一致的，它们能够通过本地地址（localhost）访问其他用户容器的端口。

在Kubernetes网络模型中，每一个Pod都拥有一个扁平化共享网络命名空间的IP，称为PodIP。通过PodIP，Pod就能够跨网络与其他物理机和容器进行通信。

在Pod运行后，我们可以查询Pod的PodIP：

```
$ kubectl get pod my-app --template={{.status.podIP}}  
10.0.10.204
```

这是一个10.0.10.0/24网段的IP，实际上这是由Docker为容器进行网络虚拟化隔离而分配的内部IP。也可以设置Pod为Host网络模式，即直接使用宿主机的网络，不进行网络虚拟化隔离。这样一来，Pod中的所有容器就直接暴露在宿主机的网络环境中，这时候，Pod的PodIP就是其所在Node的IP。

下面定义的Pod设置为Host网络模式（.spec.hostNetwork=true）：


```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - name: app
    image: nginx
    ports:
    - name: web
      containerPort: 80
      protocol: tcp
  hostNetwork: true
```

使用Host网络模式需要特别注意，一方面，因为不存在网络隔离，容易发生端口冲突；另一方面，Pod可以直接访问宿主机上的所有网络设备和服务，从安全性上来说这是不可控的。

4.5 Pod的重启策略

Pod的重启策略指的是当Pod中的容器终止退出后，重启容器的策略。需要注意的是，因为Docker容器的轻量级，重启容器的做法实际上是直接重建容器，所以容器中的数据将会丢失，如有需要持久化的数据，那么需要使用数据卷进行持久化设置。

重启策略是通过Pod定义中的.spec.restartPolicy进行设置的，目前支持以下3种策略。

- **Always:** 当容器终止退出后，总是重启容器，默认策略。
- **OnFailure:** 当容器终止异常退出（退出码非0）时，才重启容器。
- **Never:** 当容器终止退出时，从不重启容器。

现在创建一个Pod，其中的容器将异常退出（exit 1），而Pod的重启策略为OnFailure，Pod的定义文件on-failure-restart-pod.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: on-failure-restart-pod
spec:
  containers:
  - name: container
    image: ubuntu:14.04
    command: ["bash", "-c", "exit 1"]
  restartPolicy: OnFailure
```

通过定义文件创建Pod:

```
$ kubectl create -f on-failure-restart-pod.yaml
pods/on-failure-restart-pod
```

Pod创建成功后，一段时间后查询Pod:

```
$ kubectl get pod on-failure-restart-pod
```

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

on-failure-restart-pod 0/1 Error 3 2m

在Pod的查询信息中，属性RESTARTS 的值为3，说明Pod中的容器已经重启，可以分别查询每个容器的重启次数：

```
$ kubectl get pod on-failure-restart-pod \
--template="{{range .status.containerStatuses}}{{.name}}:
{{.restartCount}}{{end}}"
container:3
```

提示

关于Pod 中容器的重启次数统计，实际上并不是非常精确，只能作为一个参考。首先它获取 Pod 所在 Node 的所有容器（包括运行和停止的）作为统计基数，所以如果将退出容器进行清理（可能是人工删除，另外 Kubelet 会根据配置定时进行清理），将会减少统计的基数。

4.6 Pod的状态和生命周期

4.6.1 容器状态

Pod的本质是一组容器，Pod的状态便是容器状态的体现和概括，同时容器的状态变化会影响Pod的状态变化，触发Pod的生命周期阶段

转换。

在使用 `docker run` 运行容器的时候，首先会下载容器镜像。下载成功后运行容器，当容器运行结束退出后（包括正常和异常退出），容器终止，这是一个容器的生命周期过程。相应的，**Kubernetes** 中对于 **Pod** 中的容器进行了状态的记录，其中每种状态下包含的信息如下所示。

- **Waiting**: 容器正在等待创建，比如正在下载镜像。
 - **Reason**: 等待的原因。
- **Running**: 容器已经创建，并且正在运行。
 - **startedAt**: 容器创建时间。
- **Terminated**: 容器终止退出。
 - **exitCode**: 退出码。
 - **signal**: 容器退出信号。
 - **reason**: 容器退出原因。
 - **message**: 容器退出信息。
 - **startedAt**: 容器创建时间。
 - **finishedAt**: 容器退出时间。
 - **containerID**: 容器的ID。

Pod运行后，可以查询其中容器的状态：

```
$ kubectl describe pod my-pod
```

```
...
```

Containers:

container1:

Container ID:

docker://60387df7e5f2c781ed508084ffa3579f05482c6b465abb3d4fcae0a
de064b813

Image: nginx

Image ID:

docker://6886fb5a9b8d73b12d842bab8f9a6941c36094c2974abddb685d54c
9d99e37da

State: Running

Started: Sun, 22 Nov 2015 18:00:42 +0800

Ready: True

Restart Count: 0

Environment Variables:

...

4.6.2 Pod的生命周期阶段

Pod的生命周期可以简单描述为：首先Pod被创建，紧接着Pod被调度到Node进行部署运行。Pod是非常忠诚的，一旦被分配到Node后，就不会离开这个Node，直到它被删除，生命周期完结。

Pod的生命周期被定义为以下几个阶段。

- **Pending**: Pod已经被创建，但是一个或者多个容器还未创建，这包括Pod调度阶段，以及容器镜像的下载过程。

- **Running**: Pod已经被调度到Node，所有容器已经创建，并且至少一个容器在运行或者正在重启。

- **Succeeded**: Pod中所有容器正常退出。

- **Failed**: Pod中所有容器退出，至少有一个容器是一次退出的。

可以查询Pod处于生命周期的哪个阶段：

```
$ kubectl get pods my-app --template="{.status.phase}"
```

Running

Pod被创建成功后，首先会进入**Pending**阶段，然后被调度到Node后运行，进入**Running**阶段。如果Pod中的容器停止（正常或者异常退出），那么Pod根据重启策略的不同会进入不同的阶段，举例如下。

- Pod是**Running**阶段，含有一个容器，容器正常退出：

如果重启策略是**Always**，那么会重启容器，Pod保持**Running**阶段。

如果重启策略是**OnFailure**，Pod进入**Succeeded**阶段。

如果重启策略是**Never**，Pod进入**Succeeded**阶段。

- Pod是**Running**阶段，含有一个容器，容器异常退出：

如果重启策略是**Always**，那么会重启容器，Pod保持**Running**阶段。

如果重启策略是**OnFailure**，Pod保持**Running**阶段。

如果重启策略是Never，Pod进入Failed阶段。

- Pod是Running阶段，含有两个容器，其中一个容器异常退出：

如果重启策略是Always，那么会重启容器，Pod保持Running阶段。

如果重启策略是OnFailure，Pod保持Running阶段。

如果重启策略是Never，Pod保持Running阶段。

- Pod是Running阶段，含有两个容器，两个容器都异常退出：

如果重启策略是Always，那么会重启容器，Pod保持Running阶段。

如果重启策略是OnFailure，Pod保持Running阶段。

如果重启策略是Never，Pod进入Failed阶段。

一旦被分配到Node，Pod就不会离开这个Node，直到被删除。删除可能是人为地删除，或者被Replication Controller删除，也有可能是当Pod进入Succeeded或者Failed阶段过期，被Kubernetes清理掉。总之Pod被删除后，Pod的生命周期就算结束，即使被Replication Controller进行重建，那也是新的Pod，因为Pod的ID已经发生了变化，所以实际上Pod迁移，准确的说法是在新的Node上重建Pod。

4.6.3 生命周期回调函数

Kubernetes提供了回调函数，在容器的生命周期的特定阶段执行调用，比如容器在停止前希望执行某项操作，就可以注册相应的钩子函数。目前提供的生命周期回调函数如下所示。

- **PostStart:** 在容器创建成功后调用该回调函数。
- **PreStop:** 在容器被终止前调用该回调函数。

钩子函数的实现方式有以下两种。

• **Exec**

说明

在容器中执行指定的命令。

配置参数

command: 需要执行的命令，字符串数组。

示例

```
exec:  
  command:  
    - cat  
    - /tmp/health
```

• **HTTP**

说明

发起一个HTTP调用请求。

配置参数

path: 请求的URL路径，可选项。

port: 请求的端口，必选项。

host: 请求的IP，可选项，默认是Pod的PodIP。

scheme: 请求的协议，可选项，默认是HTTP。

示例

```
httpGet:  
  host: 192.168.1.1  
  path: /notify  
  port: 8080
```

现在定义一个Pod，包含一个Java的Web应用容器，其中设置了PostStart和PreStop回调函数。即在容器创建成功后，复制/sample.war到/app目录。而在容器被终止之前，发送HTTP请求到http://monitor.com:8080/warning，往监控系统发送一个警告，Pod的定义如下：

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: javaweb-2  
spec:
```

```
containers:
- image: resouer/sample:v2
  name: war
  lifecycle:
    postStart:
      exec:
        command:
          - "cp"
          - "/sample.war"
          - "/app"
    preStop:
      httpGet:
        host: monitor.com
        path: /warning
        port: 8080
        scheme: HTTP
```

4.7 自定义检查Pod

对于Pod是否健康，即Pod中的容器是否健康，默认情况下只是检查容器是否正常运行。但有时候容器正常运行不代表应用健康，有可能应用的进程已经阻塞住无法正常处理请求，所以为了提供更加健壮的应用，往往需要定制化的健康检查。

除此之外，有的应用启动后需要进行一系列初始化处理，在初始化完成之前应用是无法正常处理请求的。如果应用初始化需要较长时间，而实际上容器创建的时间是可以忽略不计的。默认情况下，

Kubernetes发现容器创建成功并运行，就会认为其准备就绪，真实情况是容器里的应用可能还处于初始化阶段，无法正常接受请求。如果用户访问就会得到错误响应，这不是我们希望看到的情况。同样的，我们需要更加精确的检查机制来判断Pod和容器是否准备就绪，从而让Kubernetes判断是否分发请求给Pod。

针对这些需求，Kubernetes中提供了Probe机制，有以下两种类型的Probe。

- **Liveness Probe**: 用于容器的自定义健康检查，如果Liveness Probe检查失败，Kubernetes将杀死容器，然后根据Pod的重启策略来决定是否重启容器。

- **Readiness Probe**: 用于容器的自定义准备状况检查，如果Readiness Probe检查失败，Kubernetes将会把Pod从服务代理的分发后端移除，即不会分发请求给该Pod。

Probe支持以下三种检查方法。

- **ExecAction**

说明

在容器中执行指定的命令进行检查，当命令执行成功（返回码为0），检查成功。

配置参数

command: 检查的命令，字符串数组。

示例

```
exec:  
  command:  
    - cat  
    - /tmp/health
```

• TCPSocketAction

说明

对于容器中的指定TCP端口进行检查，当TCP端口被占用，检查成功。

配置参数

port: 检查的TCP端口

示例

```
tcpSocket:  
  port: 8080
```

• HTTPGetAction

说明

发生一个HTTP请求，当返回码介于200~400之间时，检查成功。

配置参数

path: 请求的URI路径，可选项。

port: 请求的端口，必选项。

host: 请求的IP，可选项，默认是Pod的PodIP。

scheme: 请求的协议，可选项，默认是HTTP。

示例

```
httpGet:
  path: /healthz
  port: 8080
```

4.7.1 Pod的健康检查

定义一个Pod，使用Liveness Probe通过ExecAction方式检查容器的健康状态，Pod的定义文件liveness-exec-pod.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-exec-pod
```

```

labels:
  test: liveness
spec:
  containers:
  - name: liveness
    image: "ubuntu:14.04"
    command:
    - /bin/sh
    - -c
    - echo ok > /tmp/health; sleep 60; rm -rf /tmp/health; sleep
600
  livenessProbe:
    exec:
      command:
      - cat
      - /tmp/health
    initialDelaySeconds: 15
    timeoutSeconds: 1

```

通过定义文件创建Pod:

```

$ kubectl create -f liveness-exec-pod.yaml
pod "liveness-exec-pod" created

```

Pod创建之初运行正常:

```

$ kubectl get pod liveness-exec-pod

```

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

```
liveness-exec-pod    1/1          Running    0          15s
```

过1分钟以后可以看到Pod发生了重启:

```
$ kubectl get pod liveness-exec
```

NAME	READY	STATUS	RESTARTS	AGE
liveness-exec-pod	1/1	Running	1	1m

通过查询Pod事件可以看到, Liveness Probe检查失败:

```
$ kubectl describe pod liveness-exec-pod|grep Unhealthy
```

```
... Unhealthy    Liveness probe failed: cat: /tmp/health: No such
file or directory
```

4.7.2 Pod的准备状况检查

定义一个Pod, 使用Readiness Probe通过ExecAction方式检查容器的准备状况, Pod的定义文件readiness-exec-pod.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: readiness
  name: readiness-exec-pod
spec:
  containers:
  - name: readiness
```

```

image: "ubuntu:14.04"
command:
- /bin/sh
- -c
- echo ok > /tmp/ready; sleep 60; rm -rf /tmp/ready; sleep
600
readinessProbe:
  exec:
    command:
    - cat
    - /tmp/ready
  initialDelaySeconds: 15
  timeoutSeconds: 1

```

通过定义文件创建Pod:

```

$ kubectl create -f readiness-exec-pod.yaml
pod "readiness-exec-pod" created

```

Pod创建之初运行正常，容器全部准备就绪:

```

$ kubectl get pod readiness-exec

```

NAME	READY	STATUS	RESTARTS	AGE
readiness-exec-pod	1/1	Running	0	26s

过1分钟以后，发现Pod的READY数目变为0:

```

$ kubectl get pod readiness-exec-pod

```

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----


```
readiness-exec-pod    0/1          Running    0          1m
```

通过查询Pod事件可以看到，Readiness Probe检查失败：

```
$ kubectl describe pod readiness-exec | grep Unhealthy
```

```
... Unhealthy    Readiness probe failed: cat: /tmp/ready: No such
file or directory
```

4.8 调度Pod

Pod的调度指的是Pod在创建之后分配到哪一个Node上，调度算法分为两个步骤，第一步筛选出符合条件的Node，第二步选择最优的Node。

对于所有Node，首先Kubernetes通过一系列过滤函数，去除不符合条件的Node，当前版本（Kubernetes v1.1.1）支持的过滤函数如下所示。

- **NoDiskConflict**: 检查Pod请求的数据卷是否与Node上已存在Pod挂载的数据卷存在冲突，如果存在冲突，则过滤掉该Node。

- **PodFitsResources**: 检查Node的可用资源（CPU和内存）是否满足Pod的资源请求。

- **PodFitsPorts**: 检查Pod设置的HostPorts在Node上是否已经被其他Pod占用。

- **PodFitsHost**: 如果Pod设置了NodeName属性，则筛选出指定的Node。

- PodSelectorMatches: 如果Pod设置了NodeSelector属性, 则筛选出符合的Node。

- CheckNodeLabelPresence : 检查 Node 是否存在 Kubernetes Scheduler配置的标签。

筛选出符合条件的Node来运行Pod, 如果存在多个符合条件的Node, 那么需要选择出最优的Node。Kubernetes中通过一系列优先级函数(Priority Function)来评估出最优Node。对于每个Node, 优先级函数给出一个分数: 0~10 (10表示最优, 0表示最差), 而每个优先级函数设置有权重值, Node的最终分数就是每个优先级函数给出的分数进行加权的和, 比如有两个优先级函数priorityFunc1和priorityFunc2, 它们的权重值分别是weight1和weight2, 那么对于NodeA的最终分数是:

$$\text{finalScoreNodeA} = (\text{weight1} * \text{priorityFunc1}) + (\text{weight2} * \text{priorityFunc2})$$

这样一来, 通过最终分数对Node进行排序, 得分最高的Node即最优Node。如果存在多个Node并列第一, 则随机选择一个Node。

当前版本(Kubernetes v1.1.1)的Kubernetes提供的优先级函数有如下几个。

- LeastRequestedPriority: 优先选择有最多可用资源的Node。
- CalculateNodeLabelPriority: 优先选择含有指定Label的Node。
- BalancedResourceAllocation: 优先选择资源使用均衡的Node。

如何进行Node选择呢?

在一些场景下希望Pod调度到指定的Node上，比如有的Node专门用于测试；Pod在正式上线前，需要先在测试的Node上运行，测试完成再发布到生产环境的Node上运行。这时候就可以用到Node Selector，通过Node的Label进行选择。

查询所有的Node:

```
$ kubectl get node
```

NAME		LABELS	
STATUS	AGE		
kube-node-1	kubernetes.io/hostname=kube-node-1	Ready	8d
kube-node-2	kubernetes.io/hostname=kube-node-2	Ready	8d
kube-node-3	kubernetes.io/hostname=kube-node-3	Ready	8d

目前共有3个Node，状态都是Ready，并且有一个默认的Label kubernetes.io/hostname，然后为Node kube-node-1增加新的Label:

```
$ kubectl label nodes kube-node-1 env=test
```

```
node "kube-node-1" labeled
```

在定义Pod的时候通过设置 Node Selector（.spec.nodeSelector）来选择 Node，Pod的定义文件nginx-pod.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
```

```
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    env: test
```

Pod创建成功后将会被分配到Node kube-node-1:

```
$ kubectl get pod nginx -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	NODE
nginx	1/1	Running	0	9s	kube-node-1

除了设置 Node Selector 之外， Pod 还可以通过 Node Name (.spec.nodeName) 直接指定Node:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
```

```
imagePullPolicy: IfNotPresent
nodeName: kube-node-1
```

不过还是建议使用Node Selector，因为通过Label进行选择是一种弱绑定，而直接指定Node Name是强绑定，Node失效时会导致Pod无法调度。

4.9 问题定位指南

Pod是应用的承载体，当Pod运行异常的时候，可能是Kubernetes系统问题，也可能是应用本身的问题，那么就需要提供足够的信息用于问题定位，Kubernetes针对Pod提供的事件记录、日志查询和远程调试功能进行问题定位。

4.9.1 事件查询

Kubernetes从Pod的创建开始，在Pod的生命周期内会产生各种事件信息，比如Pod完成调度、下载镜像完成等。在Pod运行异常的时候，通过排除相关事件可以了解是否是由于Kubernetes的原因导致Pod异常。

事件查询可以先查询所有的事件：

```
$ kubectl get event
```

然后再查询Pod相关的事件：

```
$ kubectl describe pod my-pod
```

4.9.2 日志查询

日志是一项很重要的信息，可以用来定位问题和显示应用运行状态。Docker容器可以使用`docker logs`命令查询日志，可以通过`kubectl logs`命令查询Pod中容器的日志。

现在要定义一个Pod，包含两个容器，容器`container1`输出一条日志然后正常退出（`exit 0`），容器`container2`输出一条日志异常退出（`exit 1`），并且设置Pod的重启策略是`OnFailure`，即当容器异常退出时才进行重启，Pod的定义文件`log-pod.yaml`：

```
apiVersion: v1
kind: Pod
metadata:
  name: log-pod
spec:
  containers:
    - name: container1
      image: ubuntu:14.04
      command:
        - "bash"
        - "-c"
        - "echo \"container1: `date --rfc-3339 ns`; exit 0"
    - name: container2
      image: ubuntu:14.04
```

```
command:
- "bash"
- "-c"
- "echo \"container2: `date --rfc-3339 ns`; exit 1"
restartPolicy: OnFailure
```

通过定义文件创建Pod:

```
$ kubectl create -f log-pod.yaml
pod "log-pod" created
```

Pod创建成功后，会重新创建异常退出的容器container2:

```
$ kubectl get pod log-pod
```

NAME	READY	STATUS	RESTARTS	AGE
log-pod	0/2	Error	1	19s

然后分别查询Pod中两个容器的日志:

```
$ kubectl logs log-pod container1
container1: 2015-11-21 14:52:55.622701243+00:00
```

```
$ kubectl logs log-pod container2
Pod "log-pod" in namespace "default": container "container2" is
in waiting state.
```

因为容器container2将会异常退出然后重建，所以将处于异常状态，从而查询不到当前运行日志。但是kubectl logs可以查询之前容器

（如果存在的话）的日志，这对于问题定位非常有帮助，往往容器停止前的日志价值更高，获取方法只需要加上`--previous/-p`参数：

```
$ kubectl logs log-pod container2 --previous
```

```
container2: 2015-11-21 14:53:37.377629086+00:00
```

4.9.3 Pod的临终遗言

前面我们提到过容器停止前的日志价值更高，能够获取最后的错误异常消息、调用栈等，我们可以把这些信息形象地称为临终遗言，临终遗言对于问题定位是很有帮助的。在Kubernetes中为Pod提供了一个持久化文件，用来保存临终遗言。

Pod的定义中通过`.spec.containers[].terminationMessagePath`指定在容器中的临终遗言日志文件的路径，默认值是`/dev/termination-log`。这个文件在Pod的整个生命周期内都会保存，每次新建一个Pod，都会在宿主机上创建一个文件，然后挂载到Pod的容器中，这些文件不会因为容器的销毁而丢失，所以容器可以把临终遗言写入这个文件，方便问题排错。

现在创建一个Pod，其中的容器将写入临终遗言，Pod的定义文件`w-message-pod.yaml`：

```
apiVersion: v1
kind: Pod
metadata:
  name: w-message-pod
```



```
spec:
  containers:
  - name: messenger
    image: "ubuntu:14.04"
    terminationMessagePath: /dev/termination-log
    command:
    - "bash"
    - "-c"
      - "echo \"`date --rfc-3339 ns` I was going to die\" >>
/dev/termination-log;"
```

通过定义文件创建Pod:

```
$ kubectl create -f w-message-pod.yaml
```

```
pod "w-message-pod" created
```

```
$ kubectl get pod w-message-pod
```

NAME	READY	STATUS	RESTARTS	AGE
w-message-pod	0/1	Running	2	3m

Pod运行后，容器将往文件/dev/termination-log写入临终遗言，然后可以进行查询:

```
$ kubectl get pod w-message-pod \
--template="{{range .status.containerStatuses}}
{{.lastState.terminated.message}}{{end}}"
2015-11-21 15:11:57.457833141+00:00 I was going to die
```

4.9.4 远程连接容器

问题定位时往往需要连接到应用的运行环境进行操作，相比于传统的SSH方式，Docker提供了`docker attach`和`docker exec`两个命令可以连接容器进行操作。同样的，Kubernetes对应地提供了`kubectl attach`和`kubectl exec`两个命令用来远程连接Pod中的容器。

其中`attach`命令使用起来不太方便，相比之下，`exec`命令则非常强大，我们可以使用`kubectl exec`命令远程连接Pod中的容器运行命令（当Pod只有一个容器时，不需要指定容器）：

```
$ kubectl exec my-pod -- date
Wed Jan  6 18:19:07 CST 2016
```

或者直接进入Pod的容器中：

```
$ kubectl exec -ti my-pod /bin/bash
[root@ my-pod /]#
```

提示

`kubectl exec`命令需要在Kubernetes Node上安装nsenter。

第5章

Replication Controller

Replication Controller是Kubernetes中的另一个核心概念，应用托管在Kubernetes之后，Kubernetes需要保证应用能够持续运行，这是Replication Controller的工作内容，它会确保任何时候Kubernetes中有指定数量的Pod副本（或者称为实例）。在此基础上，Replication Controller进一步提供了高级特性，比如弹性伸缩、滚动升级等。本章将详细介绍Replication Controller，其中也会涉及一些相关的内容。

5.1 持续运行的Pod

Kubernetes 提供 Replication Controller 来管理 Pod，Replication Controller确保任何时候Kubernetes集群中有指定数量的Pod副本在运行。如果少于指定数量的Pod副本，Replication Controller会启动新的Pod，反之会杀死多余的以保证数量不变。我们通过Replication Controller来创建持续运行的Pod，Replication Controller的定义文件my-nginx-rc.yaml如下所示：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
```

```
replicas: 2
selector:
  app: nginx
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx
        ports:
          - containerPort: 80
```

定义文件中描述了Replication Controller的属性和行为，其中的主要要素如下所示。

- **apiVersion**: 声明Kubernetes的API版本，目前是v1。

- **kind**: 声明API对象的类型，这里的类型是Replication Controller。

- **metadata**: 设置Replication Controller的元数据。

- **name**: 指定Replication Controller的名称，名称必须在Namespace内唯一。

- **spec**: 配置Replication Controller的具体规格。

- **replicas**: 设置Replication Controller控制的Pod的副本数目。

- selector: 指定Replication Controller的Label Selector来匹配Pod的Label。

- template: 设置Pod模板，同Pod的定义一致。

通过定义文件创建Replication Controller:

```
$ kubectl create -f my-nginx-rc.yaml
replicationcontroller "my-nginx" created
```

查询Replication Controller:

```
$ kubectl get replicationcontroller my-nginx
```

CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR	REPLICAS
my-nginx	nginx	nginx	app=nginx	2

AGE
14s

同时可以查询到Replication Controller创建的Pod:

```
$ kubectl get pod --selector app=nginx --label-columns app
```

NAME	READY	STATUS	RESTARTS	AGE	APP
my-nginx-14lrs	1/1	Running	0	42s	nginx
my-nginx-cz2gd	1/1	Running	0	42s	nginx

因为Replication Controller设置Pod的副本数目（.spec.replicas）为2，所以创建出两个Pod，并且可以看出Pod的名称前缀是Replication Controller的名称，后缀则是5位随机字符串。

现在删除一个Pod:

```
$ kubectl delete pod my-nginx-14lrs
```

```
pod "my-nginx-14lrs" deleted
```

马上可以看到一个新的Pod被创建:

```
$ kubectl get pod --selector app=nginx
```

NAME	READY	STATUS	RESTARTS	AGE
my-nginx-cz2gd	1/1	Running	0	2m
my-nginx-vc43t	1/1	Running	0	29s

最后删除Replication Controller:

```
$ kubectl delete rc my-nginx
```

```
replicationcontroller "my-nginx" deleted
```

相应的Pod也会被删除:

```
$ kubectl get pods --selector app=nginx
```

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

使用 `kubectl delete` 命令删除 Replication Controller，默认会删除 Replication Controller 关联的所有 Pod 副本。如果需要保留 Pod 运行，删除 Replication Controller 的时候可以设置参数 `--cascade=false`。

除了 `kubectl create` 命令之外，也可以通过 `kubectl run` 命令创建 Replication Controller，下面的命令将创建名称为 `my-nginx` 的 Replication Controller，创建成功返回 Replication Controller 的信息:

```
$ kubectl run my-nginx --image nginx --replicas 2 --labels  
app=nginx
```

```
replicationcontroller "my-nginx" created
```

5.2 Pod模板

在实际操作中，相比于直接创建Pod，一般都是在Replication Controller中预先定义Pod模板，通过Replication Controller来创建Pod。

Pod模板是在Replication Controller的定义中通过.spec.template设置的。Pod模板的定义方法和Pod的定义一致，但是有一些需要注意的内容。

- 在Pod模板中不需要指定Pod的名称，如果指定了，不会报错但是不起作用。因为通过Pod模板创建Pod的时候会设置Pod的.metadata.generateName，然后Pod的名称就是.metadata.generateName拼接上5位随机码，这样做的目的是为了通过Pod模板创建出来的Pod的名称是唯一的。

- Pod模板中的重启策略必须是Always，因为Replication Controller要保证Pod持续运行，必须要求Pod总是重启容器，不然谈何持续运行。

- Pod模板中的Label不能为空，否则Replication Controller无法同Pod模板创建出来的Pod进行关联。

下面是一个Pod模板示例：

```
template:
  metadata:
```

```
labels:
  app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

Replication Controller使用Pod模板创建Pod，一旦创建成功，Pod模板和创建的Pod就没有关系了。可以修改Pod模板但不会对已创建的Pod有任何影响，也可以直接更新通过Replication Controller创建的Pod。

提示

在Kubernetes中可以单独创建Pod模板，PodTemplate的定义文件podtemplate.yaml:

```
apiVersion: v1
kind: PodTemplate
metadata:
  name: nginx
template:
  metadata:
    labels:
      name: nginx
  spec:
```



```
containers:
  - image: nginx
    name: nginx
    ports:
      - containerPort: 80
```

通过定义文件创建PodTemplate:

```
$ kubectl create -f podtemplate.yaml
```

```
podtemplate "nginx" created
```

```
$ kubectl get podTemplate nginx
```

TEMPLATE	CONTAINER(S)	IMAGE(S)	PODLABELS
nginx	nginx	nginx	name=nginx

另外，在Replication Controller定义中可通过.spec.templateRef引用PodTemplate（这部分暂未实现）：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 2
  templateRef:
    name: nginx
```

5.3 Replication Controller和Pod的关联

Replication Controller和Pod的关联就是通过Label实现的，Label机制是Kubernetes中很重要的一个设计，通过Label进行对象的弱关联，可以灵活地进行分类和选择。就像在社交网络上，对每个人打上不同的标签：职业、年龄、兴趣爱好等，然后系统可以根据标签推送不同的业务，以提供灵活的定制服务。

对于Pod，需要设置Label来进行标识，Label是一系列的Key/Value对，Pod（或者Pod模板）的定义中通过.metadata.labels设置：

```
labels:  
  key1: value1  
  key2: value2
```

Label的定义是任意的，但是Label必须具有可标识性，比如设置Pod的应用名称和版本号等。另外，Label是不具有唯一性的，为了更准确地标识Pod，建议为Pod设置不同维度的Label，比如：

- "release" : "stable", "release" : "canary"
- "environment" : "dev", "environment" : "qa", "environment" : "production"
- "tier" : "frontend", "tier" : "backend", "tier" : "cache"
- "partition" : "customerA", "partition" : "customerB"

- "track" : "daily", "track" : "weekly"

对于Replication Controller来说就是通过Label Selector来匹配Pod的Label，从而实现关联关系。在Replication Controller的定义中通过.spec.selector来设置Label Selector，Replication Controller的定义文件my-web-rc.yaml:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-web
spec:
  selector:
    app: my-web
  template:
    metadata:
      labels:
        app: my-web
        version: v1
    spec:
      containers:
        - name: my-web
          image: my-web:v1
          ports:
            - containerPort: 80
```

通过定义文件创建Replication Controller:

```
$ kubectl create -f my-web-rc.yaml
replicationcontroller "my-web" created
```

```
$ kubectl get replicationcontroller my-web
```

CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR	REPLICAS
my-web	my-web	my-web:v1	app=my-web	1

AGE
3s

通过Label查询Replication Controller创建出来的Pod:

```
$ kubectl get pod --selector app=my-web --label-columns=app
```

NAME	READY	STATUS	RESTARTS	AGE	APP
my-web-itaxw	1/1	Running	0	27s	my-web

可将该Pod的Label app=my-web修改掉:

```
$ kubectl label pod my-web-itaxw app=debug --overwrite
```

```
pod "my-web-itaxw" labeled
```

通过查询可以看到马上有新的Pod被创建出来，因为Replication Controller正是通过Label关联Pod，Pod的Label被修改掉，对Replication Controller而言相当于减少一个关联的Pod，自然就会创建新的Pod。

```
$ kubectl get pod --selector app --label-columns=app
```

NAME	READY	STATUS	RESTARTS	AGE	APP
my-web-itaxw	1/1	Running	0	1m	debug
my-web-p41ln	1/1	Running	0	16s	my-web

对于修改标签的Pod来说，相当于脱离Replication Controller的控制，这种方法适合对应用进行调试（或者数据备份），相当于有一个脱离控制的Pod可以进行调试，最后可以删除这个Pod：

```
$ kubectl delete pod my-web-itaxw
```

```
pod "my-web-itaxw" deleted
```

```
$ kubectl get pod --selector app --label-columns=app
```

NAME	READY	STATUS	RESTARTS	AGE	APP
my-web-p41ln	1/1	Running	0	1m	my-web

如果试图创建Label为app=my-web的Pod，会发现Pod是创建不出来的，因为这个Pod和Replication Controller的Label Selector匹配了，那么Replication Controller会删除这个Pod来保证Pod的副本数不多也不少。所以当多个Replication Controller的Label Selector一样且设置的Pod副本不一样的时候，会产生冲突。

5.4 弹性伸缩

弹性伸缩是指适应负载变化，以弹性可伸缩方式提供资源，特别是在虚拟化支持下，提高资源的利用率和用户满意度，较好地解决了资源利用率和应用系统之间的矛盾，是云计算领域的关键能力。想象一下现实世界，比如大型超市的出口，高峰时期人流量大的时候，适时增加结算柜台，而当人流量少的时候减少结算柜台。

同样反映到Kubernetes中，可根据负载的高低动态调整Pod的副本数，目前Kubernetes提供了API接口实现Pod的弹性伸缩。当然，Pod的

副本数本来就是通过Replication Controller进行控制，所以Pod的弹性伸缩就是修改Replication Controller的Pod副本数，可以通过kubectl scale命令来完成。

首先创建Replication Controller，设置的Pod副本数为1，Replication Controller的定义文件my-nginx-rc.yaml:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 1
  selector:
    app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
      restartPolicy: Never
```

通过定义文件创建Replication Controller:

```
$ kubectl create -f my-nginx-rc.yaml
replicationcontroller "my-nginx" created
```

Replication Controller创建成功后，创建出指定数目的Pod:

```
$ kubectl get replicationcontroller my-nginx
```

CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR	REPLICAS	AGE
my-nginx	nginx	nginx	app=nginx	1	1m

```
$ kubectl get pod --selector app=nginx
```

NAME	READY	STATUS	RESTARTS	AGE
my-nginx-sb6my	1/1	Running	0	1m

扩容Pod的副本数目到3:

```
$ kubectl scale replicationcontroller my-nginx --replicas=3
replicationcontroller "my-nginx" scaled
```

```
$ kubectl get replicationcontroller --selector app=nginx
```

CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR	REPLICAS	AGE
my-nginx	nginx	nginx	app=nginx	3	2m

```
$ kubectl get pod --selector app=nginx
```

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

my-nginx-2hzis	1/1	Running	0	49s
my-nginx-7b66n	1/1	Running	0	49s
my-nginx-sb6my	1/1	Running	0	2m

缩容Pod的副本数到1:

```
$ kubectl scale replicationcontroller my-nginx --replicas=1
replicationcontroller "my-nginx" scaled
```

```
$ kubectl get replicationcontroller --selector app=nginx
```

CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR	REPLICAS	AGE
my-nginx	nginx	nginx	app=nginx	1	3m

```
$ kubectl get pod --selector app=nginx
```

NAME	READY	STATUS	RESTARTS	AGE
my-nginx-7b66n	1/1	Running	0	1m

kubectl scale如果设置--current-replicas参数，会先检查当前的Pod的副本数是否匹配，不匹配的话会报错:

```
$ kubectl scale rc my-nginx --current-replicas=2 --replicas=1
Expected replicas to be 2, was 1
```

另外，也可以把Pod的副本数目设置为0，即删除Replication Controller关联的所有Pod:


```
$ kubectl scale replicationcontroller my-nginx --replicas=0
replicationcontroller "my-nginx" scaled
```

```
$ kubectl get replicationcontroller --selector app=nginx
```

CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR	REPLICAS	AGE
my-nginx	nginx	nginx	app=nginx	0	4m

```
$ kubectl get pod --selector app=nginx
```

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

5.5 自动伸缩

通过Replication Controller可以非常方便地实现Pod的弹性伸缩，在此基础上，只要有平台监控支持，就可以实现自动伸缩的功能，即基于Pod的资源使用情况，根据配置的策略自动调整Pod的副本数。

在Kubernetes中通过Horizontal Pod Autoscaler来实现Pod的自动伸缩，Horizontal Pod Autoscaler同Replication Controller是一一对应的，

Horizontal Pod Autoscaler将定时从平台监控系统中获取Replication Controller关联Pod的整体资源使用情况。当策略匹配的时候，通过Replication Controller来调整Pod的副本数，实现自动伸缩。

提示

在当前版本（Kubernetes v1.1.1）中，Horizontal Pod Autoscaler 处于Beta测试阶段，目前策略只支持根据CPU的使用情况进行关联，并且Kubernetes需要安装平台监控（可参考2.3.2节）。

通过kubect1 run创建Replication Controller，将运行一个Nginx Pod:

```
$ kubect1 run nginx --image=nginx --labels app=nginx --requests  
cpu=200m
```

```
replicationcontroller "nginx" created
```

```
$ kubect1 get replicationcontroller nginx
```

CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR	REPLICAS
AGE				
nginx	nginx	nginx	app=nginx	1
1m				

```
$ kubect1 get pod --selector app=nginx
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-dqsvi	1/1	Running	0	1m

通过kubect1 expose创建Service，代理 Nginx Pod，然后就可以通过 http://10.254.133.192访问Nginx Pod:

```
$ kubect1 expose replicationcontroller nginx --port=80
```

```
service "nginx" exposed
```

```
$ kubect1 get service nginx
```

NAME	CLUSTER_IP	EXTERNAL_IP	PORT(S)	SELECTOR
AGE				
nginx	10.254.133.192	<none>	80/TCP	app=nginx
5s				

现在将创建Horizontal Pod Autoscaler来实现Nginx Pod的弹性伸缩，Horizontal Pod Autoscaler定义文件nginx-hpa.yaml:

```
apiVersion: extensions/v1beta1
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
  namespace: default
spec:
  scaleRef:
    kind: ReplicationController
    name: nginx
    subresource: scale
  minReplicas: 1
  maxReplicas: 10
  cpuUtilization:
    targetPercentage: 50
```

在此Horizontal Pod Autoscaler中通过.spec.scaleRef指定对应的ReplicationController，.spec.minReplicas和.spec.maxReplicas分别设定Pod伸缩的最小和最大副本数。另外设置了自动伸缩策略：当所有关联Pod的CPU平均使用率超过50%的时候进行扩容，而少于50%的时候，进行缩容。

现在通过定义文件创建Horizontal Pod Autoscaler:

```
$ kubectl create -f nginx-hpa.yaml
horizontalpodautoscaler "nginx" created
```

提示

可以通过kubectl autoscale创建Horizontal Pod Autoscaler:

```
$ kubectl autoscale rc nginx --min=1 --max=10 --cpu-percent=50
replicationcontroller "nginx" autoscaled
```

创建成功后查询Horizontal Pod Autoscaler:

```
$ kubectl get horizontalpodautoscaler nginx
```

NAME	REFERENCE	TARGET		
CURRENT	MINPODS	MAXPODS	AGE	
nginx	ReplicationController/nginx/scale	50%	0%	1
10	1d			

因为没有访问量，所以当前关联Pod的CPU平均使用率为0%，我们可以使用工具增加访问量。当访问量增加的时候，查询Horizontal Pod Autoscaler，观察变化:

```
$ kubectl get horizontalpodautoscaler nginx
```

NAME	REFERENCE	TARGET	
CURRENT	MINPODS	MAXPODS	AGE

nginx	ReplicationController/nginx/scale	50%	60%	1
10	1d			

当CPU平均使用率超过50%的时候，可以发现对应的Replication Controller的Pod副本数变为2，实现了扩容。

```
$ kubectl get horizontalpodautoscaler nginx
```

CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR	REPLICAS
AGE				
nginx	nginx	nginx	app=nginx	2
10m				

当我们停止访问，CPU平均使用率降到50%以下，Replication Controller的Pod副本数变为1，即实现了缩容。

```
$ kubectl get horizontalpodautoscaler nginx
```

NAME	REFERENCE	TARGET
CURRENT	MINPODS MAXPODS AGE	
nginx	ReplicationController/nginx/scale	50% 43% 1
10	1d	

```
$ kubectl get replicationcontroller nginx
```

CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR	REPLICAS
AGE				
nginx	nginx	nginx	app=nginx	1
12m				

5.6 滚动升级

滚动升级是一种平滑过渡的升级方式，通过逐步替换的策略，保证整体系统的稳定，在初始升级的时候就可以及时发现、调整问题，以保证问题影响度不会扩大。

在Kubernetes中支持滚动升级，现在我们通过一个例子演示应用从V1版本滚动升级到V2版本。首先创建V1版本的Replication Controller，V1版本的Replication Controller的定义文件my-web-v1-rc.yaml:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-web-v1
spec:
  selector:
    app: my-web
    version: v1
  template:
    metadata:
      labels:
        app: my-web
        version: v1
    spec:
      containers:
        - name: my-web
          image: my-web:v1
          ports:
            - containerPort: 80
              protocol: TCP
```

通过定义文件创建Replication Controller:

```
$ kubectl create -f my-web-v1-rc.yaml
```

```
replicationcontroller "my-web-v1" created
```

```
$ kubectl get replicationcontroller my-web-v1
```

CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR
REPLICAS	AGE		
my-web-v1	my-web	my-web:v1	app=my-web, version=v1
1	10s		

然后扩容Replication Controller的Pod副本数目到4:

```
$ kubectl scale rc my-web-v1 --replicas=4
```

```
replicationcontroller "my-web-v1" scaled
```

```
$ kubectl get pods --selector app=my-web
```

NAME	READY	STATUS	RESTARTS	AGE
my-web-v1-3dd6v	1/1	Running	0	9s
my-web-v1-8893c	1/1	Running	0	57s
my-web-v1-il6ii	1/1	Running	0	8s
my-web-v1-rzjp5	1/1	Running	0	9s

现在需要应用从 V1 版本升级到 V2 版本, V2 版本的 Replication Controller 定义文件my-web-v2-rc.yaml:

```
apiVersion: v1
```

```
kind: ReplicationController
```

```
metadata:
```

```
    name: my-web-v2
spec:
  selector:
    app: my-web
    version: v2
  template:
    metadata:
      labels:
        app: my-web
        version: v2
    spec:
      containers:
      - name: my-web
        image: my-web:v2
        ports:
        - containerPort: 80
          protocol: TCP
```

开始滚动升级:

```
$ kubectl rolling-update my-web-v1 -f my-web-v2-rc.yaml --  
update-period=10s
```

Created my-web-v2

Scaling up my-web-v2 from 0 to 4, scaling down my-web-v1 from 4
to 0 (keep 4 pods available,
don't exceed 5 pods)

Scaling my-web-v2 up to 1

Scaling my-web-v1 down to 3


```
Scaling my-web-v2 up to 2
Scaling my-web-v1 down to 2
Scaling my-web-v2 up to 3
Scaling my-web-v1 down to 1
Scaling my-web-v2 up to 4
Scaling my-web-v1 down to 0
Update succeeded. Deleting my-web-v1
replicationcontroller "my-web-v1" rolling updated to "my-web-v2"
```

升级开始后，首先根据提供的定义文件创建V2版本的Replication Controller，然后每隔10s（通过kubectrl rolling-update的参数--update-period设置）逐步增加V2版本的Replication Controller的Pod副本数，逐步减少V1版本的Replication Controller的Pod副本数。升级完成后删除V1版本的Replication Controller，保留V2版本的Replication Controller，即实现滚动升级。

```
Updating my-web-v1 replicas: 3, my-web-v2 replicas: 1
Updating my-web-v1 replicas: 2, my-web-v2 replicas: 2
Updating my-web-v1 replicas: 1, my-web-v2 replicas: 3
Updating my-web-v1 replicas: 0, my-web-v2 replicas: 4
```

升级期间通过查询Pod可知，V2版本的Pod正在逐渐替换V1版本的Pod，当然这是通过调整Replication Controller的副本数目来控制的，下面显示的是升级过程中的一个阶段的状况：

```
$ kubectl get replicationcontroller --selector app=my-web
CONTROLLER          CONTAINER(S)        IMAGE(S)             SELECTOR
REPLICAS   AGE
```

my-web-v1	my-web	my-web:v1	app=my-web, version=v1
2	2m		
my-web-v2	my-web	my-web:v2	app=my-web, version=v2
2	48s		

\$ kubectl get pod --selector app=my-web

NAME	READY	STATUS	RESTARTS	AGE
my-web-v1-3dd6v	1/1	Running	0	1m
my-web-v1-8893c	1/1	Running	0	2m
my-web-v2-6cg74	1/1	Running	0	19s
my-web-v2-b4qaz	1/1	Running	0	45s

待升级完成，即V2版本的Pod完全替换V1版本的Pod，同时V1版本的Replication Controller也被V2版本的Replication Controller替换：

\$ kubectl get replicationcontroller --selector app=my-web

CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR
my-web-v2	my-web	my-web:v2	app=my-web , version=v2
4			

\$ kubectl get pod --selector app=my-web

NAME	READY	STATUS	RESTARTS	AGE
my-web-v2-785ok	1/1	Running	0	35s
my-web-v2-bg8ur	1/1	Running	0	1m
my-web-v2-qr33c	1/1	Running	0	1m
my-web-v2-y2es3	1/1	Running	0	57s

如果在升级过程中，发生了错误中途退出的时候，可以选择继续升级。Kubernetes能够智能地判断出升级中断之前的阶段，然后紧接着继续执行升级。另外，也可以进行回退，命令如下：

```
$ kubectl rolling-update my-web-v1 -f my-web-v2-rc.yaml --  
update-period=10s --rollback
```

```
Setting "my-web-v1" replicas to 4
```

```
Continuing update with existing controller my-web-v1.
```

```
Scaling up my-web-v1 from 3 to 4, scaling down my-web-v2 from 2  
to 0 (keep 2 pods available,  
don't exceed 3 pods)
```

```
Scaling my-web-v2 down to 0
```

```
Scaling my-web-v1 up to 4
```

```
Update succeeded. Deleting my-web-v2
```

回退的方式实际上就是升级的逆操作，逐步增加V1版本的Replication Controller的副本数，逐步减少V2版本的Replication Controller的副本数。

5.7 Deployment

Kubernetes提供了一种更加简单的更新Replication Controller和Pod的机制，叫作Deployment。

提示

在当前版本（Kubernetes v1.1.1）中，Deployment处于beta测试阶段，需要Kubernetes API Server的启动参数设置`--runtime-config=extensions/v1beta1/deployments=true`开启Deployment支持。

我们创建一个Deployment，Deployment的定义文件my-web-v1-deployment.yaml:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-web-deployment
spec:
  replicas: 4
  template:
    metadata:
      labels:
        app: my-web
    spec:
      containers:
        - name: my-web
          image: my-web:v1
          ports:
            - containerPort: 80
              protocol: TCP
```

Deployment的定义方法与Replication Controller类似，包括Pod副本数和Pod副本的设置，这些定义将会作用于Deployment创建的

Replication Controller。

通过定义文件创建Deployment:

```
$ kubectl create -f my-web-v1-deployment.yaml --validate=false
deployment "my-web-deployment" created
```

创建成功后可以查询到 Deployment 和其创建的 Replication Controller:

```
$ kubectl get deployment my-web-deployment
```

NAME	UPDATED	REPLICAS	AGE
my-web-deployment	0/4		32s

```
$ kubectl get replicationcontroller --selector app=my-web
```

CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR
deploymentrc-3379117081	my-web	my-web:v1	app=my-web,
deployment.kubernetes.io/podTemplateHash=3379117081	0		21s

Deployment创建出来的Replication Controller的名称是deploymentrc-3379117081，Replication Controller使用的是Deployment定义中的Pod模板。

Deployment 给 Replication Controller 添加了一个 Label deployment.kubernetes.io/ podTemplateHash=3379117081，其中Label的Value是3379117081，这是由Pod模板计算出的Hash值，Label的Key是由Deployment 中的 .spec.uniqueLabelKey 指定的，默认是

deployment.kubernetes.io/podTemplateHash, 如果为空, 则不添加这个 Label。

Deployment中通过.spec.replicas指定了预期的Pod副本数, 不过在刚创建的时候, 初始值是0。

过一段时间后, Deployment将会设置Replication Controller的Pod副本数为4, 相应地创建出了4个Pod:

```
$ kubectl get deployment my-web-deployment
```

NAME	UPDATEDREPLICAS	AGE
my-web-deployment	4/4	50s

```
$ kubectl get replicationcontroller --selector app=my-web
```

CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR
deploymentrc-3379117081	my-web	my-web:v1	app=my-web,
deployment.kubernetes.io/podTemplateHash=3379117081	4	1m	

```
$ kubectl get pods --selector app=my-web --label-columns
```

deployment.kubernetes.io/

podTemplateHash

NAME	READY	STATUS
deploymentrc-3379117081-5ghmj	Running	0
1m 3379117081		
deploymentrc-3379117081-g4a0i	Running	0
1m 3379117081		

deploymentrc-3379117081-orckb	1/1	Running	0
1m	3379117081		
deploymentrc-3379117081-r69kt	1/1	Running	0
1m	3379117081		

现在我们需要更新应用，镜像my-web:v1升级到my-web:v2，为此，新的Deployment定义文件my-web-v2-deployment.yaml如下所示：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-web-deployment
spec:
  replicas: 4
  template:
    metadata:
      labels:
        app: my-web
    spec:
      containers:
        - name: my-web
          image: my-web:v2
          ports:
            - containerPort: 80
              protocol: TCP
```

使用新的定义文件更新Deployment:

```
$ kubectl apply -f my-web-v2-deployment.yaml --validate=false
deployment "my-web-deployment" configured
```

更新生效后，可以查询到 Deployment 创建了新的 Replication Controller:

```
$ kubectl get replicationcontroller --selector app=my-web
```

CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR
deploymentrc-3379117081	my-web	my-web:v1	app=my-web , deployment.kubernetes.io/podTemplateHash=3379117081
deploymentrc-3445177370	my-web	my-web:v2	app=my-web , deployment.kubernetes.io/podTemplateHash=3445177370

REPLICAS AGE

4 2m

0 6s

新创建出的 Replication Controller 名称为 deploymentrc-3445177370，Label 设置为 deployment.kubernetes.io/podTemplateHash=3445177370，3445177370 是由新的 Pod 模板计算出的 Hash 值，新的 Replication Controller 使用镜像 my-web:v2，初始的 Pod 副本数为 0。

紧接着 Deployment 就会控制新旧 Replication Controller，实现 Pod 的升级:

```
$ kubectl get replicationcontroller --selector app=my-web
```

CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR
deploymentrc-3379117081	my-web	my-web:v1	app=my-web , deployment.kubernetes.io/podTemplateHash=3379117081
deploymentrc-3445177370	my-web	my-web:v2	app=my-web , deployment.kubernetes.io/podTemplateHash=3445177370

REPLICAS AGE

0 4m

4 2m


```
$ kubectl get pod --selector app=my-web --label-columns  
deployment.kubernetes.io/
```

```
podTemplateHash
```

NAME	READY	STATUS
RESTARTS AGE PODTEMPLATEHASH		
deploymentrc-3445177370-4ej3a 1/1	Running	0
2m 3445177370		
deploymentrc-3445177370-i327a 1/1	Running	0
1m 3445177370		
deploymentrc-3445177370-reeqh 1/1	Running	0
46s 3445177370		
deploymentrc-3445177370-zhopb 1/1	Running	0
1m 3445177370		

默认情况下，Deployment采取的是滚动升级方式，同使用kubectl rolling-update是一致的，即逐步增加新的Replication Controller的副本数，逐步减少旧的Replication Controller的副本数，通过查询Deployment的事件可以看到具体情况：

```
$ kubectl describe deployment my-web-deployment
```

```
...
```

```
Scaled up rc deploymentrc-3379117081 to 4
```

```
Scaled up rc deploymentrc-3445177370 to 1
```

```
Scaled down rc deploymentrc-3379117081 to 2
```

```
Scaled up rc deploymentrc-3445177370 to 3
```

```
Scaled down rc deploymentrc-3379117081 to 0
```

```
Scaled up rc deploymentrc-3445177370 to 4
```

在Deployment中可以配置升级的策略，通过.spec.strategy.type指定升级类型，包括Recreate和RollingUpdate。

- Recreate：直接升级，即删除所有旧的Pod，然后创建新的Pod，当前版本（Kubernetes v1.1.1）暂未实现。
- RollingUpdate：滚动升级，支持参数配置，如表5-1所示。

表5-1 RollingUpdate策略参数

参数	说明
.spec.strategy.rollingUpdate.maxUnavailable	允许的最大失效Pod数值，可选配置，值可以是绝对值（5）或者是比例（10%），默认值是1。比如maxUnavailable是30%，升级开始后，旧的Replication Controller可以立即缩容30%的Pod，当新的Replication Controller创建出Pod以后，旧的Replication Controller可以进一步缩容，整个升级过程至少需要保证70%的Pod可用
.spec.strategy.rollingUpdate.maxSurge	允许超过指定Pod数目

的最大数值，可选配置，值可以是绝对值（5）或者是比例（10%），默认值是1。比如maxSurge是30%，升级开始后，新的Replication Controller可以立即扩容30%，旧的Replication Controller删除Pod之后，新的Replication Controller可以继续扩容，在整个过程中，新旧Pod的总数目不可以超过指定数目的130%

5.8 一次性任务的Pod

从程序运行形态上来区分，我们可以将Pod分为两类：长时运行服务和一次性任务。大部分应用比如Nginx、Redis等都是长时运行服务，另外也存在一次性任务的场景，比如执行冒烟测试、数据计算等。

Replication Controller创建的Pod都是长时运行服务，相应的，Kubernetes提供了另一种机制，Job来管理一次性任务的Pod。

提示:

在当前版本（Kubernetes v1.1.1）中，Job处于Beta测试阶段。

我们现在使用Job来计算圆周率，Job的定义文件pi-job.yaml:

```
apiVersion: extensions/v1beta1
kind: Job
metadata:
  name: pi
spec:
  completions: 1
  parallelism: 1
  selector:
    matchLabels:
      app: pi
  template:
    metadata:
      name: pi
      labels:
        app: pi
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle", "print
```

```
bpi(2000)"]
    restartPolicy: Never
```

Job同样是通过.spec.template配置Pod的模板的，因为是一次性任务Pod，所以Pod的重启策略只能是Never或者OnFailure。

Job可以控制一次性任务Pod的完成次数（.spec.completions）和并发执行数（.spec.parallelism），当Pod成功执行指定次数后即认为Job执行完毕。

通过定义文件创建Job:

```
$ kubectl create -f pi-job.yaml
job "pi" created
```

```
$ kubectl get job
```

JOB	CONTAINER(S)	IMAGE(S)	SELECTOR	SUCCESSFUL
pi	pi	perl	app in (pi)	0

Job创建成功后，将会创建运行Pod:

```
$ kubectl get pod --selector app=pi
```

NAME	READY	STATUS	RESTARTS	AGE
pi-8117p	1/1	Running	0	6s

Pod是一次性任务，计算出圆周率就终止，我们可以查询到Pod输出的圆周率:

```
$ kubectl logs pi-8117p
```

```
3.14159265358979323846264338327950288419716939937510582097494459
```

2307816406286208998628034...

在一次性任务Pod执行完后，显示Job已经成功执行1次，即完成任务：

```
$ kubectl get job pi
```

JOB	CONTAINER(S)	IMAGE(S)	SELECTOR	SUCCESSFUL
pi	pi	perl	app in (pi)	1

第6章

Service

为了适应快速的业务需求，微服务架构已经逐渐成为主流，微服务架构的应用需要有非常好的服务编排支持。Kubernetes中的核心要素Service便提供了一套简化的服务代理和发现机制，天然适应微服务架构，任何应用都可以非常轻易地运行在Kubernetes中而无须对架构进行改动。本章将阐述Service的基本概念和功能细节，最后介绍如何将Service发布给外部网络的方法。

6.1 Service代理Pod

在Kubernetes中，在受到Replication Controller支配的时候，Pod副本是变化的，比如发生迁移（准确说是Pod的重建）或者伸缩的时候。这对于Pod的访问者来说就是一种负担，访问者需要能够发现这些Pod副本，并且感知Pod副本的变化以便及时进行更新。

Kubernetes中的Service是一种抽象概念，它定义了一个Pod逻辑集合以及访问它们的策略，Service同Pod的关联同样是居于Label来完成的。Service的目标是提供一种桥梁，它会为访问者提供一个固定访问地址，用于在访问时重定向到相应的后端，这使得非Kubernetes原生应用程序，在无须为Kubernetes编写特定代码的前提下，轻松访问后端。

我们现在定义一个 Service 来代理 Pod，首先创建 Replication Controller，Replication Controller的定义文件my-nginx-rc.yaml:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

通过定义文件创建Replication Controller:

```
$ kubectl create -f my-nginx-rc.yaml
replicationcontroller "my-nginx" created
```

Replication Controller创建出3个Pod副本:


```
$ kubectl get pod --selector app=nginx
```

NAME	READY	STATUS	RESTARTS	AGE
my-nginx-2ywz1	1/1	Running	0	1m
my-nginx-arjq1	1/1	Running	0	1m
my-nginx-tok0x	1/1	Running	0	1m

然后创建Service，Service的定义文件my-nginx-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
spec:
  selector:
    app: nginx
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
```

定义文件中描述了Service的属性和行为，其中的主要要素如下所示

- apiVersion: 声明Kubernetes的API版本，目前是v1。
- kind: 声明API对象的类型，这里类型是Service。
- metadata: 设置Service的元数据。

- name: 指定Service的名称，名称必须在Namespace内唯一。
- spec: 配置Service的具体规格。
 - selector: 指定Service的Label Selector来匹配Pod的Label。
 - ports: 设置Service的端口转发规则。

通过定义文件创建Service:

```
$ kubectl create -f my-nginx-service.yaml
service "my-nginx" created
```

提示

除了 kubectl create 之外，也可以通过 kubectl expose 创建 Service:

```
$ kubectl expose replicationcontroller my-nginx --name=my-
nginx --port=80 --target-
port=80
service "my-nginx" exposed
```

创建成功后可以查询Service:

```
$ kubectl get service my-nginx
```

NAME	CLUSTER_IP	EXTERNAL_IP	PORT(S)	SELECTOR
AGE				

```
my-nginx    10.254.226.117    <none>      80/TCP      app=nginx
43s
```

Service同Replication Controller一样都是通过Label来关联Pod的，Service的定义中指定了Label Selector为app=nginx，那么就会关联前面运行的3个Pod。Service会将关联到的Pod作为Service分发请求的后端，可以查询Service：

```
$ kubectl describe service my-nginx
```

```
Name:      my-nginx
Namespace:  default
Labels:     <none>
Selector:   app=nginx
Type:       ClusterIP
IP:         10.254.226.117
Port:       <unnamed> 80/TCP
Endpoints:  10.0.62.68:80, 10.0.62.69:80, 10.0.62.70:80
Session Affinity: None
No events.
```

可以看到Service的Endpoints属性包含了3个IP: 10.0.62.68、10.0.62.69、10.0.62.70，实际上这就是Service关联的3个Pod的PodIP：

```
$ kubectl get pod --selector app=nginx -o yaml|grep podIP
```

```
podIP: 10.0.62.70
podIP: 10.0.62.68
podIP: 10.0.62.69
```

我们看到，Service的IP属性显示为10.254.226.117，实际上这是Kubernetes分配给Service的一个虚拟IP。通过访问Service的虚拟IP，Kubernetes会转发请求到后端Pod。另外，Service的端口转发规则显示Service的80/TCP端口（通过spec.ports[0].port指定）转发到后端的80端口（通过spec.ports[0].targetPort指定），比如访问10.254.226.117:80的请求会被转发到后端10.0.62.68:80、10.0.62.69:80、10.0.62.70:80：

```
$ curl 10.254.226.117:80
...Thank you for using nginx...
```

而当Pod发生变化的时候，Service会及时更新，比如将Pod的副本数目减少至1：

```
$ kubectl scale replicationcontroller my-nginx --replicas=1
replicationcontroller "my-nginx" scaled
```

```
$ kubectl get pod --selector app=nginx
```

NAME	READY	STATUS	RESTARTS	AGE
my-nginx-tok0x	1/1	Running	0	7m

Service相应地进行了更新：

```
$ kubectl describe service my-nginx
```

```
Name:      my-nginx
Namespace:  default
Labels:    <none>
Selector:  app=nginx
Type:      ClusterIP
```

```
IP:      10.254.226.117
Port:    <unnamed> 80/TCP
Endpoints: 10.0.62.69:80
Session Affinity: None
No events.
```

这样一来，Service就可以作为Pod的访问入口，起到代理服务器的作用，而对于访问者来说，通过Service进行访问，无须直接感知Pod。

6.2 Service的虚拟IP

Kubernetes分配给Service一个固定IP，这是一个虚拟IP（也称为ClusterIP），并不是一个真实存在的IP，而是由Kubernetes虚拟出来的。虚拟IP的范围通过Kubernetes API Server的启动参数--service-cluster-ip-range=10.254.0.0/16配置，查询Service：

```
$ kubectl get service
```

NAME	CLUSTER_IP	EXTERNAL_IP	PORT(S)
SELECTOR	AGE		
kubernetes	10.254.0.1	<none>	443/TCP
<none>	6d		
my-nginx	10.254.226.117	<none>	80/TCP
app=nginx	43s		

可以查询到两个Service，其中第一个Service是由Kubernetes默认创建的，它代表着Kubernetes API Server。两个Service的虚拟IP都属于

10.254.0.0/16范围，在Service定义中可以通过.spec.clusterIP指定虚拟IP，指定的IP必须在指定范围内，并且该虚拟IP未被分配使用：

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
spec:
  selector:
    app: nginx
  ports:
    - name: http
      port: 80
      targetPort: 80
      protocol: TCP
  clusterIP: 10.254.249.161
```

如果 Service 设置.spec.clusterIP 为 None，表示不给 Service 分配虚拟 IP，我们称为Headless Service：

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
spec:
  selector:
    app: nginx
  ports:
```

```
- name: http
  port: 80
  targetPort: 80
  protocol: TCP
clusterIP: None
```

虚拟IP属于Kubernetes内部的虚拟网络，外部是寻址不到的。在Kubernetes系统中，实际上是由Kubernetes Proxy组件负责实现虚拟IP路由和转发的，所以在Kubernetes Node中我们都运行了Kubernetes Proxy，从而在容器覆盖网络之上又实现了Kubernetes层级的虚拟转发网络。

6.3 服务代理

在逻辑层面上，Service可以被认为是真实应用的抽象，每一个Service关联着一系列的Pod。在物理层面上，Service又是真实应用的代理服务器，对外表现为一个单一访问入口，通过Kubernetes Proxy转发请求到Service关联的Pod。

Service同样是根据Label Selector来筛选Pod进行关联的，实际上Kubernetes在Service和Pod之间通过Endpoints衔接，Endpoints同Service关联的Pod相对应，可以认为是Service的服务代理后端，Kubernetes会根据Service关联到的Pod的PodIP信息组合成一个Endpoints。

我们现在创建一个Service，定义文件my-nginx-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
spec:
  selector:
    app: nginx
  ports:
    - name: http
      port: 80
      targetPort: 80
      protocol: TCP
    - name: https
      port: 443
      targetPort: 443
      protocol: TCP
```

`Service` 的定义中设置了两个端口转发规则。当 `Service` 只配置一个端口的时候，端口的名称是可选项，而当 `Service` 配置多个端口的时候，每个端口的 `name` 就是必选项。

通过定义文件创建 `Service`:

```
$ kubectl create -f my-nginx-service.yaml
```

```
service "my-nginx" created
```

```
$ kubectl get service my-nginx
```

NAME	CLUSTER_IP	EXTERNAL_IP	PORT(S)
------	------------	-------------	---------

SELECTOR	AGE		
my-nginx	10.254.181.218	<none>	80/TCP , 443/TCP
app=nginx	3s		

Service将通过Label app=nginx关联到1个Pod:

```
$ kubectl get pod --selector app=nginx
```

NAME	READY	STATUS	RESTARTS	AGE
my-nginx-4s97i	1/1	Running	0	48s

Kubernetes 创建 Service 的同时，会自动创建跟 Service 同名的 Endpoints:

```
$ kubectl get endpoints my-nginx -o yaml
```

```
apiVersion: v1
```

```
kind: Endpoints
```

```
metadata:
```

```
  creationTimestamp: 2015-11-28T03:35:52Z
```

```
  name: my-nginx
```

```
  namespace: default
```

```
  resourceVersion: "224471"
```

```
  selfLink: /api/v1/namespaces/default/endpoints/my-nginx
```

```
  uid: 1ff974f4-9581-11e5-b92e-005056817c3e
```

```
subsets:
```

```
- addresses:
```

```
  - ip: 10.0.62.71
```

```
    targetRef:
```

```
      kind: Pod
```

```
    name: my-nginx-4s97i
    namespace: default
    resourceVersion: "224453"
    uid: 33e8b6e0-9581-11e5-b92e-005056817c3e
  ports:
  - name: http
    port: 80
    protocol: TCP
  - name: https
    port: 443
    protocol: TCP
```

可以看到Endpoints包含Service关联到的Pod的PodIP，Service根据端口对应Endpoints的相应端口：

```
$ kubectl describe service my-nginx
```

```
Name:    my-nginx
Namespace: default
Labels:   <none>
Selector: app=nginx
Type:     ClusterIP
IP:       10.254.181.218
Port:     http 80/TCP
Endpoints: 10.0.62.71:80
Port:     https 443/TCP
Endpoints: 10.0.62.71:443
```

Session Affinity: None

No events.

Service不仅可以代理Pod，还可以代理任意其他后端，比如运行在Kubernetes外部的服务。假设现在要使用一个Service代理外部MySQL服务，不用设置Service的Label Selector，Service的定义文件mysql-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  ports:
    - port: 3036
      targetPort: 3036
      protocol: TCP
```

同时定义跟Service同名的Endpoints，Endpoints中设置了MySQL的IP: 192.168.3.180，Endpoints的定义文件mysql-endpoints.yaml:

```
apiVersion: v1
kind: Endpoints
metadata:
  name: mysql
subsets:
  - addresses:
      - ip: 192.168.3.180
```

```
ports:
- port: 3036
  protocol: TCP
```

通过定义文件创建Service和Endpoints:

```
$ kubectl create -f mysql-service.yaml -f mysql-endpoints.yaml
service "mysql" created
endpoints "mysql" created
```

可以查询到Service将指向我们自定义的Endpoints:

```
$ kubectl get endpoints mysql
```

NAME	ENDPOINTS	AGE
mysql	192.168.3.180:3036	23s

```
$ kubectl describe service mysql
```

```
Name:      mysql
Namespace: default
Labels:    <none>
Selector:  <none>
Type:      ClusterIP
IP:        10.254.223.0
Port:      <unnamed> 3036/TCP
Endpoints: 192.168.3.180:3036
Session Affinity: None
No events.
```

当Service的Endpoints包含多个IP的时候，即服务代理存在多个后端，将进行请求的负载均衡，默认的负载均衡策略是轮询或者随机（根据Kubernetes Proxy的模式决定）。Service支持基于源IP地址的会话保持，通过.spec.sessionAffinity设置为ClientIP:

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
spec:
  selector:
    app: nginx
  ports:
    - name: http
      port: 80
      targetPort: 80
      protocol: TCP
  sessionAffinity: ClientIP
```

6.4 服务发现

微服务架构是一种新流行的架构模式，相比于传统的单块架构模式，微服务架构提倡将应用划分成一组小的服务。每个服务运行在其独立的进程中，服务之间互相协调、互相配合，服务与服务间采用轻量级的通信机制互相沟通。每个服务都围绕着具体业务进行构建，并且能够被独立部署和运行。

可以说Docker的轻量级容器技术天然适合微服务架构，每一个微服务都通过Docker进行打包、部署和运行。而Docker的集装箱能力为微服务架构保驾护航，可以快速、轻松地实现每个组件的测试和升级，将微服务架构的优势最大化。

但是应用的微服务化也将带来新的挑战，其中一个就是把应用划分成多个分布式组件运行，每个组件又将进行集群化扩展，组件和组件之间的相互发现和通信将会变得复杂起来，而一套服务编排机制就显得非常重要。

Kubernetes提供了强大的服务编排能力，微服务化应用的每一个组件都以Service进行抽象，组件和组件之间只需要访问Service即可以互相通信，而无须感知组件的集群变化。同时Kubernetes为Service提供了服务发现的能力，组件和组件之间可以简单地互相发现。

Kubernetes中支持两种服务发现方法：环境变量和DNS。

6.4.1 环境变量

当有Pod被创建的时候，Kubernetes将为Pod设置每一个Service的相关环境变量，这些环境变量包括两种类型。

• Kubernetes Service环境变量

Kubernetes为Service设置的环境变量形式，包括：

`{SVCNAME}_SERVICE_HOST`

`{SVCNAME}_SERVICE_PORT`

`{SVCNAME}_SERVICE_PORT_{PORTNAME}`

其中的服务名和端口名转为大写，连字符转换为下画线。

• Docker Link环境变量

相当于通过Docker的--link参数实现容器连接时设置的环境变量形式，可参考<https://docs.docker.com/userguide/dockerlinks/>。

比如现在有一个Service，查询信息如下：

```
$ kubectl describe service my-dns
```

```
Name:      my-dns
```

```
Namespace: default
```

```
Labels:     <none>
```

```
Selector:   run=my-dns
```

```
Type:       ClusterIP
```

```
IP:         10.254.105.183
```

```
Port:       dns    53/UDP
```

```
Endpoints:  10.0.10.24:53
```

```
Port:       dns-tcp 53/TCP
```

```
Endpoints:  10.0.10.24:53
```

```
Session Affinity: None
```

```
No events.
```

这个Service对应的环境变量如下：

```
# Kubernetes Service环境变量
MY_DNS_SERVICE_HOST=10.254.105.183
MY_DNS_SERVICE_PORT_DNS=53
MY_DNS_SERVICE_PORT_DNS_TCP=53
MY_DNS_SERVICE_PORT=53

# Docker Link环境变量
MY_DNS_PORT_53_UDP_PORT=53
MY_DNS_PORT_53_TCP_ADDR=10.254.105.183
MY_DNS_PORT=udp://10.254.105.183:53
MY_DNS_PORT_53_UDP=udp://10.254.105.183:53
MY_DNS_PORT_53_UDP_PROTO=udp
MY_DNS_PORT_53_UDP_ADDR=10.254.105.183
MY_DNS_PORT_53_TCP_PORT=53
MY_DNS_PORT_53_TCP=tcp://10.254.105.183:53
MY_DNS_PORT_53_TCP_PROTO=tcp
```

可以看到，环境变量中记录了Service的虚拟IP以及端口和协议信息。这样一来，Pod中的程序就可以使用这些环境变量发现Service。

环境变量服务发现方式是Kubernetes默认支持的，但是此种方式存在限制。首先环境变量是租户隔离的，即Pod只能获取同Namespace中的Service的环境变量。另外，Pod和Service的创建顺序是有要求的，即Service必须在Pod创建之前被创建，否则Service环境变量不会设置到Pod中。DNS服务发现方式则没有这些限制。

6.4.2 DNS

DNS服务发现方式需要Kubernetes提供Cluster DNS支持，Cluster DNS会监控Kubernetes API，为每一个Service创建DNS记录用于域名解析，这样在Pod中就可以通过DNS域名获取Service的访问地址。而对于一个Service，Cluster DNS会创建两条DNS记录：

```
[service_name].[namespace_name].[cluster_domain]  
[service_name].[namespace_name].svc.[cluster_domain]
```

Cluster DNS的安装方法可参考2.3.1节，本书使用的Cluster DNS Server的IP为10.254.10.2，Cluster DNS的本地域为cluster.local。

比如有一个 Service 的名称是 my-service，Namespace 是 my-ns，Cluster DNS 会创建DNS记录：

```
my-service.my-ns.cluster.local  
my-service.my-ns.svc.cluster.local
```

对于如何通过Cluster DNS进行Service的域名解析，需要了解Linux的DNS域名解析机制。Linux系统中的DNS域名解析是通过/etc/resolv.conf进行配置的，它的格式很简单，每行以一个关键字开头，后接配置参数。/etc/resolv.conf中的相关关键字说明如表6-1所示。

表6-1 /etc/resolv.conf关键字

关键字	说明
nameserver	DNS服务器的IP地址，可以有很多行的nameserver，每一个带一个IP地址，在查询时就按nameserver在本文件中的顺序进行
domain	主机的域名，当为没有域名的主机进行DNS查询时使用
search	DNS服务器搜索域，它的多个参数指明域名查询顺序。当要查询没有域名的主机时，主机将在由search声明的域中分别查找
options	DNS解析选项值，以Key/Value对的方式出现

Pod中的容器使用容器宿主机的DNS域名解析配置，称为默认DNS配置。另外，如果Kubernetes部署并设置了Cluster DNS支持，那么在创建Pod的时候，默认会将Cluster DNS的配置写入Pod中容器的DNS域名解析配置中，称为Cluster DNS配置。

比如，Kubernetes Node的/etc/resolv.conf配置如下：

```
# Generated by NetworkManager
nameserver 218.85.157.99
```

在Pod的定义中通过.spec.dnsPolicy设置Pod的DNS策略，默认值是ClusterFirst，查看DNS策略设置为ClusterFirst的Pod中容器的/etc/resolv.conf：

```
$ kubectl exec my-app -- cat /etc/resolv.conf
nameserver 10.254.10.2
nameserver 218.85.157.99
search          default.svc.cluster.local      svc.cluster.local
cluster.local
options ndots:5
```

其中nameserver 10.254.10.2是Cluster DNS配置，nameserver 218.85.157.99是容器宿主机的默认DNS配置。根据先后顺序，会优先

使用 Cluster DNS 进行域名解析。另外，配置中包括根据 Pod 的 Namespace 和 Cluster Domain 设置 DNS 服务器搜索域：

```
search          [namespace_name].svc.[cluster_domain]      svc.  
[cluster_domain] [cluster_domain]
```

比如在 Namespace my-ns 下的 Pod 的 DNS 服务器的搜索域：

```
search my-ns.svc.cluster.local svc.cluster.local cluster.local
```

因为设置了 DNS 服务器的搜索域，在 Pod 中就可以使用 [service_name].[namespace_name] 访问到任意 Namespace 的 Service，而使用 [service_name] 可以访问到同 Namespace 下的 Service。比如对于 Namespace my-ns 下的 Service my-service，任意 Namespace 的 Pod 通过 my-service.my-ns 可以解析发现 Service：

```
$ kubectl exec my-pod -- nslookup my-service.my-ns --  
namespace=default
```

```
Server:  10.254.10.2
```

```
Address: 10.254.10.2#53
```

```
Name: my-service.my-ns.svc.cluster.local
```

```
Address:  10.254.0.235
```

同 Namespace 下的 Pod 可以通过 my-service 解析发现 Service：

```
$ kubectl exec my-pod -- nslookup my-service --namespace=my-ns
```

```
Server:  10.254.10.2
```

```
Address: 10.254.10.2#53
```

Name: my-service.my-ns.svc.cluster.local

Address: 10.254.0.235

另外，Cluster DNS会为Headless Service的域名添加其Endpoints的所有IP，即可以实现DNS的负载均衡：

```
$ kubectl describe service my-service
```

Name: my-service

Namespace: default

Labels: <none>

Selector: app=nginx

Type: ClusterIP

IP: None

Port: <unnamed> 80/TCP

Endpoints: 10.0.62.19:80, 10.0.62.20:80, 10.0.62.21:80

Session Affinity: None

No events.

```
$ kubectl exec my-pod -- nslookup my-service
```

Server: 10.254.10.2

Address: 10.254.10.2#53

Name: my-service.default.svc.cluster.local

Address: 10.0.62.20

Name: my-service.default.svc.cluster.local

Address: 10.0.62.19

Name: my-service.default.svc.cluster.local

Address: 10.0.62.21

6.5 发布Service

Service的虚拟IP是由Kubernetes虚拟出来的内部网络，外部网络是无法寻址到的，但是有一些Service又需要对外暴露，比如Web前端。这时候就需要增加一层网络转发，即外网到内网的转发，Kubernetes提供了NodePort Service、LoadBalancer Service和Ingress可以发布Service。

6.5.1 NodePort Service

NodePort Service是类型为NodePort的Service，Kubernetes除了会分配给NodePort Service一个内部的虚拟IP，另外会在每一个Node上暴露端口NodePort，外部网络可以通过[NodeIP]:[NodePort]访问到Service。

我们现在创建一个NodePort Service:

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
spec:
  selector:
    app: nginx
```

```
ports:
- name: http
  port: 80
  targetPort: 80
  protocol: TCP
type: NodePort
```

创建成功后查询NodePort Service:

```
$ kubectl describe service my-nginx
```

```
Name:      my-nginx
Namespace:  default
Labels:     <none>
Selector:   app=nginx
Type:       NodePort
IP:         10.254.38.180
Port:       http 80/TCP
NodePort:   http 32143/TCP
Endpoints:  <none>
Session Affinity: None
No events.
```

可以看到，Kubernetes给NodePort Service中每一个端口都创建了一个NodePort（http 32143/TCP），在NodePort Service定义中可以通过.spec.ports[].nodePort指定固定NodePort，NodePort的范围默认是30000~32767，可以通过Kubernetes API Server的启动参数--service-node-port-range指定范围。

NodePort Service 就可以通过 [NodeIP]:[NodePort] 访问，而当 NodeIP 是一个公网 IP 时，外部就可以访问到 NodePort Service 了。

6.5.2 LoadBalancer Service

LoadBalancer Service 是类型为 LoadBalancer 的 Service，LoadBalancer Service 是建立在 NodePort Service 集群上的，Kubernetes 会分配给 LoadBalancer Service 一个内部的虚拟 IP，并且暴露 NodePort。除此之外，Kubernetes 请求底层云平台创建一个负载均衡器，将每个 Node 作为后端，负载均衡器将转发请求到 [NodeIP]:[NodePort]。

LoadBalancer Service 需要底层云平台支持创建负载均衡器，比如 GCE，现在创建一个 LoadBalancer Service：

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
spec:
  selector:
    app: nginx
  ports:
    - name: http
      port: 80
      targetPort: 80
```

```
protocol: TCP
type: LoadBalancer
```

Kubernetes会分配给LoadBalancer Service一个内部的虚拟IP，并且暴露NodePort。进一步的，Kubernetes请求底层云平台创建一个负载均衡器，作为访问LoadBalancer Service的外部访问入口。负载均衡器由底层云平台创建提供，会包含一个LoadBalancerIP，可以认为是LoadBalancer Service的外部IP，查询LoadBalancer Service：

```
$ kubectl get svc my-nginx
```

NAME	CLUSTER_IP	EXTERNAL_IP	PORT(S)
my-nginx	10.254.147.57	78.110.25.19	80/TCP
app=nginx	4m		

其中EXTERNAL_IP:78.110.25.19就是LoadBalancer Service的外部IP。负载均衡器将每个Node作为后端，当请求78.110.25.19:80时，负载均衡器将转发请求到相应的[NodeIP]:[NodePort]，从而访问到LoadBalancer Service。

6.5.3 Ingress

Kubernetes提供了一种HTTP方式的路由转发机制，称为Ingress。Ingress的实现需要两个组件支持，Ingress Controller和HTTP代理服务器。HTTP代理服务器将会转发外部的HTTP请求到Service，而Ingress Controller则需要监控Kubernetes API，实时更新HTTP代理服务器的转发规则。

提示

在当前版本（Kubernetes v1.1.1）中，Ingress处于beta测试阶段。

我们现在创建一个Ingress，Ingress的定义文件my-ingress.yaml:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
  - host: my.example.com
    http:
      paths:
      - path: /app
        backend:
          serviceName: my-app
          servicePort: 80
```

Ingress 定义中的.spec.rules 设置了转发规则，其中配置了一条规则，当 HTTP 请求的host为my.example.com且path为/app时，转发到Service my-app的80端口。

通过定义文件创建Ingress:

```
$ kubectl create -f my-ingress.yaml
```

```
ingress "my-ingress" created
```

创建成功后可以查询Ingress:

```
$ kubectl get ingress my-ingress
```

NAME	RULE	BACKEND	ADDRESS
my-ingress	-		
	my.example.com		
	/app	my-app:80	

当Ingress创建成功后，需要Ingress Controller根据Ingress的配置，设置HTTP代理服务器的转发策略，外部通过HTTP代理服务器就可以访问到Service。

在当前版本（Kubernetes v1.1.1）中，Ingress Controller和HTTP代理服务器是作为外部组件运行的。官方提供了GCE Load-Balancer作为HTTP代理服务器，另外也可以使用HAProxy或者Nginx等开源方案。

以Nginx为例，将Nginx配置文件以模板形式编写：

```
const (  
    nginxConf = `  
events {  
    worker_connections 1024;  
}  
http {  
    {{range $ing := .Items}}  
    {{range $rule := $ing.Spec.Rules}}
```

```

server {
    listen 80;
    server_name {{$rule.Host}};
    resolver 127.0.0.1;
    {{ range $path := $rule.HTTP.Paths }}
        location {{$path.Path}} {
            proxy_pass http://{{$path.Backend.ServiceName}}:
{{$path.Backend.ServicePort}};
        }
    }
}

```

Ingress Controller_{监控}Kubernetes API:

```

for {
    rateLimiter.Accept()
    ingresses, err := ingClient.List(labels.Everything(),
fields.Everything())
    if err != nil || reflect.DeepEqual(ingresses.Items,
known.Items) {
        continue
    }
    if w, err := os.Create("/etc/nginx/nginx.conf"); err !=
nil {
        log.Fatalf("Failed to open %v: %v", nginxConf, err)
    } else if err := tmpl.Execute(w, ingresses); err != nil {

```

```
        log.Fatalf("Failed to write template %v", err)
    }
    shellOut("nginx -s reload")
}
```

最终生成的Nginx配置文件如下:

```
events {
    worker_connections 1024;
}
http {
    server {
        listen 80;
        server_name my.example.com;
        resolver 127.0.0.1;

        location /app {
            proxy_pass http://my-app:80;
        }
    }
}
```

这样一来，Nginx将作为访问入口，访问<http://my.example.com/app>的请求将会转发到<http://my-app:80>，即访问到Service。

第7章

数据卷

数据卷用于实现容器持久化数据，Kubernetes对于数据卷重新定义，提供了丰富强大的功能。本章将数据卷按照功能划分为三类：本地数据卷、网络数据卷和信息数据卷，并一一进行说明，包括使用方法、配置参数和示例。另外，其中结合示例详细介绍了Persistent Volume和Persistent Volume Claim。

7.1 Kubernetes数据卷

在Docker的设计实现中，容器中的数据是临时的，即当容器被销毁时，其中的数据将会丢失。如果需要持久化数据，需要使用Docker数据卷挂载宿主机上的文件或者目录到容器中。

在Kubernetes系统中，当Pod重建的时候，数据是会丢失的，Kubernetes也是通过数据卷来提供Pod数据的持久化的。Kubernetes数据卷是对Docker数据卷的扩展，Kubernetes数据卷是Pod级别的，可以用来实现Pod中容器的文件共享。

Kubernetes数据卷适配对接各种存储系统，提供了丰富强大的功能。Kubernetes提供了以下类型的数据卷：

- EmptyDir
- HostPath

- GCE Persistent Disk
- Aws Elastic Block Store
- NFS
- iSCSI
- Flocker
- GlusterFS
- RBD
- Git Repo
- Secret
- Persistent Volume Claim
- Downward API

7.2 本地数据卷

Kubernetes中有两种类型的数据卷，它们只能作用于本地文件系统，我们称为本地数据卷。本地数据卷中的数据只会存在于一台机器上，所以当Pod发生迁移的时候，数据便会丢失，无法满足真正的数据持久化要求。但是本地数据卷提供了其他用途，比如Pod中容器的文件共享，或者共享宿主机的文件系统。

7.2.1 EmptyDir

EmptyDir从名称上的意思看是空的目录，它是在Pod创建的时候新建的一个目录。

如果Pod配置了EmptyDir数据卷，在Pod的生命周期内都会存在，当Pod被分配到Node上的时候，会在Node上创建EmptyDir数据卷，并挂载到Pod的容器中。只要Pod存在，EmptyDir数据卷都会存在（容器删除不会导致EmptyDir数据卷丢失数据），但是如果Pod的生命周期终结（Pod被删除），EmptyDir数据卷也会被删除，并且永久丢失。

EmptyDir数据卷非常适合实现Pod中容器的文件共享。Pod的设计提供了一个很好的容器组合的模型，容器之间各司其职，通过共享文件目录来完成交互，比如可以通过一个专职日志收集容器，在每个Pod中和业务容器中进行组合，来完成日志的收集和汇总。

我们创建一个Pod，Pod中包含两个容器，容器synthetic-logger写日志到/var/log目录，而容器sidecar-log-collector负责收集/var/log目录下的日志文件，然后导出到Elasticsearch，其中的/var/log目录就是一个EmptyDir数据卷，分别挂载到两个容器中，从而实现文件共享。Pod的定义文件如下：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    example: logging-sidecar
  name: logging-sidecar-example
```

spec:

containers:

- name: synthetic-logger

image: ubuntu:14.04

command: ["bash","-c","i=\"0\"; while true; do echo
\"`hostname`: \$i \" >>

/var/log/synthetic-count.log; date --rfc-3339 ns >>

/var/log/synthetic-dates.log; sleep 4;

i=\${i+1}; done"]

volumeMounts:

- name: log-storage

mountPath: /var/log

- name: sidecar-log-collector

image: gcr.io/google_containers/fluentd-sidecar-es:1.2

resources:

limits:

cpu: 100m

memory: 200Mi

env:

- name: FILES_TO_COLLECT

value: "/var/log/synthetic-count.log /var/log/synthetic-
dates.log"

volumeMounts:

- name: log-storage

readOnly: true

mountPath: /var/log

volumes:


```
- name: log-storage
  emptyDir: {}
```

7.2.2 HostPath

HostPath数据卷允许将容器宿主机上的文件系统挂载到Pod中。如果Pod需要使用宿主机上的某些文件，可以使用HostPath数据卷。

创建一个Pod需要使用宿主机的SSL证书，可以通过创建HostPath数据卷，然后将宿主机的/etc/ssl/certs目录挂载到容器中，Pod的定义文件如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
          protocol: TCP
      volumeMounts:
        - name: ssl-certs
          mountPath: /etc/ssl/certs
          readOnly: true
```

```
volumes:
  - name: ssl-certs
    hostPath:
      path: /etc/ssl/certs
```

7.3 网络数据卷

Kubernetes提供了很多类型的数据卷以集成第三方的存储系统，包括一些非常流行的分布式文件系统，也有在IaaS平台上提供的存储支持，这些存储系统都是分布式的，通过网络共享文件系统，因此我们称这一类数据卷为网络数据卷。

网络数据卷能够满足数据的持久化需求，Pod通过配置使用网络数据卷，每次Pod创建的时候都会将存储系统的远端文件目录挂载到容器中，数据卷中的数据将被永久保存，即使Pod被删除的时候，只是除去挂载数据卷，数据卷中的数据仍然保存在存储系统中，并且当新的Pod被创建的时候，仍是挂载同样的数据卷。

7.3.1 NFS

NFS（Network File System）即网络文件系统，是FreeBSD支持的一种文件系统，它允许网络中的计算机通过TCP/IP共享资源。在NFS的应用中，本地NFS的客户端应用可以透明地读写位于远端NFS服务器上的文件，就像访问本地文件一样。

NFS数据卷的配置参数如表7-1所示。

表7-1 NFS数据卷的配置参数

参数	可选项	说明
server	否	NFS的服务端地址
path	否	NFS的共享目录路径
readOnly	是	是否只读，默认为false（即可读可写）。

示例如下：

```

apiVersion: v1
kind: Pod
metadata:
  name: nfs-web
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
      volumeMounts:
        - name: nfs
          mountPath: "/usr/share/nginx/html"

```

```
volumes:
  - name: nfs
    nfs:
      server: nfs-server.default.kube.local
      path: "/"
```

7.3.2 iSCSI

iSCSI技术是一种由IBM公司研究开发的，是一个供硬件设备使用的可以在IP的上层运行的SCSI指令集，这种指令集合可以实现在IP网络上运行SCSI协议，使其能够在诸如高速千兆以太网上进行路由选择。iSCSI技术是一种新的储存技术，该技术是将现有SCSI接口与以太网（Ethernet）技术结合，使服务器可与使用IP网络的储存装置互相交换资料。

iSCSI数据卷的配置参数如表7-2所示。

表7-2 iSCSI数据卷配置参数

参数	可选项	说明
targetPortal	否	iSCSI Target服务地址
iqn	否	iSCSI的IQN号
lun	否	iSCSI的逻辑单元号
fsType	否	文件系统类型，ext4、xfs或ntfs
readOnly	是	是否只读，默认为false（即可读可写）

示例如下：

```
apiVersion: v1
kind: Pod
```

metadata:

name: iscsipd

spec:

containers:

- image: kubernetes/pause

name: iscsipd-ro

volumeMounts:

- mountPath: /mnt/iscsipd

name: iscsipd-ro

- image: kubernetes/pause

name: iscsipd-rw

volumeMounts:

- mountPath: /mnt/iscsipd

name: iscsipd-rw

volumes:

- iscsi:

fsType: ext4

iqn: iqn.2001-04.com.example:storage.kube.sys1.xyz

lun: 0

readOnly: true

targetPortal: 10.0.2.15:3260

name: iscsipd-ro

- iscsi:

fsType: ext4

iqn: iqn.2001-04.com.example:storage.kube.sys1.xyz

lun: 1

```
targetPortal: 10.0.2.15:3260
name: iscsipd-rw
```

7.3.3 GlusterFS

GlusterFS是Scale-Out存储解决方案Gluster的核心，它是一个开源的分布式文件系统，具有强大的横向扩展能力，通过扩展能够支持数PB存储容量和处理数千客户端。GlusterFS借助TCP/IP或InfiniBand RDMA网络将物理分布的存储资源聚集在一起，使用单一全局命名空间来管理数据。GlusterFS基于可堆叠的用户空间设计，可为各种不同的数据负载提供优异的性能。

GlusterFS数据卷配置参数如表7-3所示。

表7-3 GlusterFS 数据卷配置参数

参数	可选项	说明
endpoints	否	GlusterFS服务端对应的Endpoint
path	否	GlusterFS数据卷路径
readOnly	是	是否只读，默认为false（即可读可写）

示例如下：

apiVersion: v1
kind: Endpoints
metadata:
 name: glusterfs-cluster

subsets:

- addresses:

- ip: 10.240.106.152

ports:

- port: 1

- addresses:

- ip: 10.240.79.157

ports:

- port: 1

apiVersion: v1

kind: Pod

metadata:

name: glusterfs

spec:

containers:

- image: kubernetes/pause

name: glusterfs

volumeMounts:

- mountPath: /mnt/glusterfs

name: glusterfsvol

volumes:

- glusterfs:

```
endpoints: glusterfs-cluster
path: kube_vol
readOnly: true
name: glusterfsvol
```

7.3.4 RBD (Ceph Block Device)

Ceph是开源、分布式的网络存储，同时又是文件系统。Ceph的设计目标是卓越的性能、可靠性以及可扩展性。Ceph基于可靠的、可扩展的和分布式的对象存储，通过一个分布式的集群管理元数据，符合POSIX。RBD (Rados Block Device) 是一个Linux块设备驱动，提供了一个共享网络块设备，实现与Ceph的交互。RBD在Ceph对象存储的集群上进行条带化和复制，提供可靠性、可扩展性以及块设备的访问。

Kubernetes 中支持RBD方式，RBD数据卷配置参数如表7-4所示。

表7-4 RBD数据卷配置参数

参数	可选项	说明
monitors	否	Ceph Monitors的地址
image	否	Rados 镜像名称
fsType	否	文件系统类型，ext4、xfs或ntfs
pool	是	Rados Pool名称，默认是rbd
user	是	Rados用户名，默认是admin
keyring	是	Rados钥匙圈文件的路径，默认为/etc/ceph/keyring
secretRef	是	Rados用户认证Secret，默认为空
readOnly	是	是否只读，默认为false（即可读可写）

示例如下：


```
apiVersion: v1
kind: Pod
metadata:
  name: rbd
spec:
  containers:
    - image: kubernetes/pause
      name: rbd-rw
      volumeMounts:
        - mountPath: /mnt/rbd
          name: rbdpd
  volumes:
    - name: rbdpd
      rbd:
        fsType: ext4
        image: foo
        keyring: /etc/ceph/keyring
        monitors:
          - 10.16.154.78:6789
          - 10.16.154.82:6789
          - 10.16.154.83:6789
        pool: kube
        readOnly: true
        user: admin
```

7.3.5 Flocker

Flocker是一个容器数据管理工具，作为ClusterHQ公司2014年推出的产品，Flocker主要负责Docker容器及其数据的管理。从功能方面而言，Flocker是一个数据卷管理器和多主机的Docker集群管理工具。用户可以通过它来控制数据，实现在Docker中运行数据库、队列和键值（Key/Value）存储等服务，并在应用程序中轻松使用这些服务。

Flocker数据卷配置参数如表7-5所示。

表7-5 Flocker 数据卷配置参数

参数	可选项	说明
datasetName	否	Flocker数据集名称

示例如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: flocker-web
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
      volumeMounts:
```

```

        # name must match the volume name below
        - name: www-root

        mountPath: "/usr/share/nginx/html"

volumes:
  - name: www-root

  flocker:
    datasetName: my-flocker-vol

```

7.3.6 AWS Elastic Block Store

Amazon Elastic Block Store（Amazon EBS）在AWS云中提供用于Amazon EC2实例的持久性数据的块级存储卷。如果Kubernetes运行在AWS之上，就可以非常方便地使用Amazon Elastic Block Store作为数据卷。

Amazon Elastic Block Store数据卷的配置参数如表7-6所示。

表7-6 Amazon Elastic Block Store数据卷配置参数

参数	可选项	说明
volumeID	否	EBS数据卷标识
fsType	否	文件系统类型，ext4、xfs或ntfs
partition	是	磁盘挂载分区
readOnly	是	是否只读，默认为false（即可读可写）

示例如下：

```

apiVersion: v1
kind: Pod
metadata:

```

```

    name: test-ebs
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-ebs
      name: test-volume
  volumes:
  - name: test-volume
    # This AWS EBS volume must already exist.
    awsElasticBlockStore:
      volumeID: aws://<availability-zone>/<volume-id>
      fsType: ext4

```

7.3.7 GCE Persistent Disk

GCE Persistent Disk 是 GCE 提供的持久化数据的服务，如果 Kubernetes 运行在 GCE 之上，可以使用 GCE Persistent Disk 作为数据卷。

GCE Persistent Disk 数据卷的配置参数如表 7-7 所示。

表 7-7 GCE Persistent Disk 数据卷配置参数

参数	可选项	说明
volumeID	否	EBS 数据卷标识
fsType	否	文件系统类型，ext4、xfs 或 ntfs
partition	是	磁盘挂载分区
readOnly	是	是否只读，默认为 false（即可读可写）

示例如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    # This GCE PD must already exist.
    gcePersistentDisk:
      pdName: my-data-disk
      fsType: ext4
```

7.4 Persistent Volume和Persistent Volume Claim

理解每个存储系统是一件复杂的事情，特别是对于普通用户来说，有时候并不关心各种存储实现，只希望能够安全可靠地存储数据。Kubernetes中提供了Persistent Volume和Persistent Volume Claim机

制，这是存储消费模式。Persistent Volume是由系统管理员配置创建的一个数据卷，它代表了某一类存储插件实现，可以是NFS、iSCSI等；而对于普通用户来说，通过Persistent Volume Claim可请求并获得合适的Persistent Volume，而无须感知后端的存储实现。

Persistent Volume Claim和Persistent Volume的关系其实类似于Pod和Node，Pod消费Node的资源，Persistent Volume Claim则是消费Persistent Volume的资源。Persistent Volume和Persistent Volume Claim相互关联，有着完整的生命周期管理。

• 准备

系统管理员规划并创建一系列的Persistent Volume，Persistent Volume在创建成功后处于可用状态。

• 绑定

用户创建Persistent Volume Claim来声明存储请求，包括存储大小和访问模式。Persistent Volume Claim创建成功后进入等待状态，当Kubernetes发现有新的Persistent Volume Claim创建的时候，就会根据条件查找Persistent Volume。当有Persistent Volume匹配的时候，就会将Persistent Volume Claim和Persistent Volume进行绑定，Persistent Volume和Persistent Volume Claim都进入绑定状态。

Kubernetes只会选择可用状态的Persistent Volume，并且采取最小满足策略，而当没有Persistent Volume满足需求的时候，Persistent Volume

Claim将处于等待阶段。比如现在有两个Persistent Volume可用，一个Persistent Volume的容量是50Gi，一个Persistent Volume的容量是60Gi。那么请求40Gi的Persistent Volume Claim会被绑定到50Gi的Persistent Volume，而请求100Gi的Persistent Volume Claim则处于等待状态，直到有大于100Gi的Persistent Volume出现（Persistent Volume有可能被新建或者被回收）。

• 使用

创建Pod的时候使用Persistent Volume Claim，Kubernetes便会查询其绑定的Persistent Volume，去调用真正的存储实现，然后将数据卷挂载到Pod中。

• 释放

当用户删除Persistent Volume所绑定的Persistent Volume Claim时，Persistent Volume则进入释放状态。此时Persistent Volume中可能残留着之前Persistent Volume Claim使用的数据，所以Persistent Volume并不可用，需要对Persistent Volume进行回收操作。

• 回收

释放的Persistent Volume需要回收才能再次使用，回收的策略可以是人工处理，或者由Kubernetes自动进行清理，如清理失败，

Persistent Volume会进入失败状态。

综上所述，Persistent Volume的状态包括如下几项。

- Available：Persistent Volume创建成功后进入可用状态，等待Persistent Volume Claim消费。

- Bound：Persistent Volume被分配到Persistent Volume Claim进行绑定，Persistent Volume进入绑定状态。

- Released：Persistent Volume绑定的Persistent Volume Claim被删除，Persistent Volume进入释放状态，等待回收处理。

- Failed：Persistent Volume执行自动清理回收策略失败后，Persistent Volume会进入失败状态。

Persistent Volume Claim的状态包括如下几项。

- Pending：Persistent Volume Claim创建成功后进入等待状态，等待绑定Persistent Volume。

- Bound：分配Persistent Volume给Persistent Volume Claim进行绑定，Persistent Volume Claim进入绑定状态。

7.4.1 创建Persistent Volume

Persistent Volume代表可靠的存储系统，从实现上来说，Persistent Volume实际上就是复用了已有的数据卷实现，配置方法也是一致的，Persistent Volume目前支持以下类型：

- GCE Persistent Disk
- AWS Elastic Block Store
- NFS
- iSCSI
- RBD (Ceph Block Device)
- GlusterFS
- HostPath: 只适合测试使用

Persistent Volume是需要事先创建好的，这一般来说是系统管理员的工作。系统管理员根据实际情况，创建一系列可用的Persistent Volume。比如Kubernetes是运行在AWS之上的，购买了10个100Gi的Amazon EBS，那就可以创建10个容量为100Gi的Persistent Volume。

创建Persistent Volume的时候需要指定数据卷的容量、访问模式和回收策略。

• 容量

Persistent Volume通过设置资源容量，然后Persistent Volume Claim在请求的时候指定资源需求来进行匹配。比如有1个容量为10Gi的Persistent Volume和1个容量为20Gi的Persistent Volume，Persistent Volume Claim请求15Gi，就匹配了20Gi的Persistent Volume。目前Persistent Volume的资源容量只有一个属性，就是存储大小：

capacity:

storage: 20Gi

• 访问模式

- **ReadWriteOnce**: 数据卷能够在单个节点上挂载为读写目录。
- **ReadOnlyMany**: 数据卷能够在多个节点上挂载为只读目录。
- **ReadWriteMany**: 数据卷能够在多个节点上挂载为读写目录。

• 回收策略

当前支持的回收策略如下所示。

- **Retain**: **Persistent Volume**在释放后，需要人工进行回收操作。
- **Recycle**: **Persistent Volume**在释放后，**Kubernetes**自动进行清理，清理成功后**Persistent Volume**则可以再次绑定使用。目前只有**NFS**和**HostPath**类型的**Persistent Volume**支持回收策略。当执行回收策略的时候，会创建一个**Persistent Volume Recycler Pod**，这个**Pod**执行清理动作，即删除**Persistent Volume**目录下的所有文件（包括隐藏文件）。

现在创建一个Persistent Volume， Persistent Volume的定义文件nfs-pv.yaml:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv
  labels:
    type: nfs
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: nfs-server
    path: /
```

Persistent Volume的定义中使用了NFS，配置参数同NFS数据卷一致，另外设置了存储容量为5Gi，访问权限为ReadWriteMany。

通过定义文件创建Persistent Volume:

```
$ kubectl create -f nfs-pv.yaml
persistentvolume "nfs-pv" created
```

创建成功后可以查询创建的Persistent Volume， Persistent Volume的状态为Available:

```
$ kubectl describe persistentvolume nfs-pv
```

```
Name:  nfs-pv
```

```
Labels:  type=nfs
```

```
Status:  Available
```

```
Claim:
```

```
Reclaim Policy: Retain
```

```
Access Modes: RWX
```

```
Capacity: 5Gi
```

```
Message:
```

```
Source:
```

```
    Type: NFS (an NFS mount that lasts the lifetime of a pod)
```

```
    Server:nfs-server
```

```
    Path: /
```

```
    ReadOnly:false
```

7.4.2 创建Persistent Volume Claim

Persistent Volume Claim指定所需要的存储大小，然后Kubernetes会选择满足条件的Persistent Volume进行绑定。

现在创建 Persistent Volume Claim 来消费刚创建的 Persistent Volume，Persistent Volume Claim的定义文件test-pvc.yaml:

```
apiVersion: v1
```

```
kind: PersistentVolumeClaim
```

```
metadata:
```

```
  name: test-pvc
```

```
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 3Gi
```

通过定义文件创建Persistent Volume Claim:

```
$ kubectl create -f my-pvc.yaml
persistentvolumeclaim "my-pvc" created
```

创建成功后可以查询创建的Persistent Volume Claim:

```
$ kubectl describe persistentvolumeclaim my-pvc
Name:      my-pvc
Namespace: default
Status:    Bound
Volume:    nfs-pv
Labels:    <none>
Capacity:  5Gi
Access Modes: RWX
```

可以看到，Persistent Volume Claim 已经绑定 Persistent Volume 为 nfs-pv，然后查询Persistent Volume的状态为Bound:

```
$ kubectl describe persistentvolume nfs-pv
Name:      nfs-pv
Labels:    type=nfs
```

```
Status:      Bound
Claim:       default/my-pvc
Reclaim Policy: Retain
Access Modes:  RWX
Capacity:            5Gi
Message:
Source:
    Type: NFS (an NFS mount that lasts the lifetime of a pod)
    Server:nfs-server
    Path: /
    ReadOnly:false
```

最后创建Pod来使用Persistent Volume Claim, Pod的定义文件:

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  labels:
    app: busybox
spec:
  containers:
  - name: busybox
    image: busybox
    command:
      - sleep
      - 3600
    volumeMounts:
```

```
- mountPath: /busybox-data
  name: data
volumes:
- name: data
  persistentVolumeClaim:
    claimName: my-pvc
```

7.5 信息数据卷

Kubernetes中有一些数据卷，主要用来给容器传递配置信息，我们称之为信息数据卷，比如Secret和Downward API，都是将Pod的信息以文件形式保存，然后以数据卷方式挂载到容器中，容器通过读取文件获取相应的信息。从功能设计上来说，是有点偏离数据卷的本意，数据卷是用来持久化数据的，或者进行文件共享的。未来版本可能会对这部分进行重构，将信息数据卷提供的功能放在更合适的地方。

7.5.1 Secret

Kubernetes提供了Secret来处理敏感数据，比如密码、Token和密钥，相比于直接将敏感数据配置在Pod的定义或者镜像中，Secret提供了更加安全的机制，防止数据泄露。

Secret的创建是独立于Pod的，以数据卷的形式挂载到Pod中，Secret的数据将以文件的形式保存，容器通过读取文件可以获取需要的数据。

Secret的类型有3种。

- Opaque: 自定义数据内容，默认值。

- kubernetes.io/service-account-token: Service Account的认证内容，可参考10.3节。

- kubernetes.io/dockercfg: Docker镜像仓库的认证内容，可参考4.3.1节。

现在有一个应用需要获取一个账号密码，即可以通过Secret来实现:

```
username: my-username
```

```
password: my-password
```

因为是自定义数据内容，所以Secret的类型是Opaque，配置的数据是一系列Key/Value对，其中Value 需要使用Base64 加密，Secret定义文件secret.yaml:

```
apiVersion: v1
```

```
kind: Secret
```

```
metadata:
```

```
  name: mysecret
```

```
type: Opaque
```

```
data:
```

```
  username: bXktdXNlcm5hbWUK
```

```
  password: bXktdGFzc3dvcmlQK
```

通过定义文件创建Secret:


```
$ kubectl create -f secret.yaml
```

```
secret "mysecret" created
```

创建成功后查询Secret:

```
$ kubectl describe secret mysecret
```

```
Name:  mysecret
```

```
Namespace:  default
```

```
Labels:  <none>
```

```
Annotations:  <none>
```

```
Type:  Opaque
```

```
Data
```

```
====
```

```
password: 12 bytes
```

```
username: 12 bytes
```

然后创建Pod使用该Secret, Pod的定义文件:

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  name: busybox
```

```
  labels:
```

```
    app: busybox
```

```
spec:
```

```
  containers:
```

```
    - name: busybox
```

```
image: busybox
command:
- sleep
- "3600"
volumeMounts:
- mountPath: /secret
  name: secret
  readOnly: true
volumes:
- name: secret
  secret:
    secretName: mysecret
```

Pod的定义中声明了Secret数据卷，并且将Secret数据卷挂载到了Pod容器中的/secret目录下，因为Secret包含两个Key/Value对，在容器的/secret目录中分别有两个文件username和password，文件的内容就是解密后的数据：

```
$ kubectl exec busybox -- ls /secret
```

```
password
```

```
username
```

```
$ kubectl exec busybox -- cat /secret/username
```

```
my-username
```

```
$ kubectl exec busybox -- cat /secret/password
```

```
my-password
```

7.5.2 Downward API

Downward API可以通过环境变量的方式告诉容器Pod的信息（可参考4.3.3节），另外，也可以通过数据卷方式传值，Pod的信息将会以文件的形式通过数据卷挂载到容器中，在容器中可以通过读取文件获取信息，目前支持：

- Pod的名称。
- Pod的Namespace。
- Pod的Label。
- Pod的Annotation。

创建Pod使用Downward API数据卷，Pod的定义文件downwardapi-volume.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: downwardapi-volume
  labels:
    zone: us-est-coast
    cluster: test-cluster1
    rack: rack-22
  annotations:
    build: two
```

```
    builder: john-doe
spec:
  containers:
    - name: client-container
      image: ubuntu:14.04
      command: ["/bin/bash", "-c", "while true; do sleep 5;
done"]
      volumeMounts:
        - name: podinfo
          mountPath: /podinfo/
          readOnly: false
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "pod_name"
            fieldRef:
              fieldPath: metadata.name
          - path: "pod_namespace"
            fieldRef:
              fieldPath: metadata.namespace
          - path: "pod_labels"
            fieldRef:
              fieldPath: metadata.labels
          - path: "pod_annotations"
            fieldRef:
              fieldPath: metadata.annotations
```

Pod定义中声明了Downward API数据卷，分别引用了Pod的相关属性，然后将数据卷挂载到容器/podinfo目录下。在Pod创建运行后，就可以查询/podinfo目录下的数据：

```
$ kubectl exec downwardapi-volume -- ls /podinfo
```

```
pod_annotations
```

```
pod_labels
```

```
pod_name
```

```
pod_namespace
```

```
$ kubectl exec downwardapi-volume -- cat /podinfo/pod_name
```

```
downwardapi-volume
```

```
$ kubectl exec downwardapi-volume -- cat /podinfo/pod_namespace
```

```
default
```

```
$ kubectl exec downwardapi-volume -- cat /podinfo/pod_labels
```

```
cluster="test-cluster1"
```

```
rack="rack-22"
```

```
$ kubectl exec downwardapi-volume -- cat
```

```
/podinfo/pod_annotations
```

```
build="two"
```

```
builder="john-doe"
```

```
kubectl.kubernetes.io/last-applied-configuration="..."
```

```
kubernetes.io/config.seen="2015-11-02T18:29:57.509960318+08:00"
```

```
kubernetes.io/config.source="api"
```

7.5.3 Git Repo

Kubernetes支持将Git仓库下载到Pod中，目前是通过Git Repo数据卷实现，即当Pod配置Git Repo数据卷时，就下载配置的Git仓库到Pod的数据卷中，然后挂载到容器中。我们现在定义一个Pod使用Git Repo数据卷，Pod的定义文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  labels:
    app: busybox
spec:
  containers:
    - name: busybox
      image: busybox
      command:
        - sleep
        - "3600"
      volumeMounts:
        - mountPath: /config
          name: busybox-config
  volumes:
    - name: busybox-config
      gitRepo:
        repository: https://github.com/wulonghui/busybox-
```

```
config.git
```

```
revision: master
```

Pod的定义中声明了Git Repo数据卷，然后挂载到容器的/config目录下，在Pod运行后，即会下载指定的Git仓库到/config目录下：

```
$ kubectl exec busybox -- ls /config
```

```
busybox-config
```

第8章

访问Kubernetes API

Kubernetes规范和健全的API模型，为Kubernetes的使用和扩展提供了非常好的支持。

本章首先说明Kubernetes提供的API对象以及元数据，然后介绍Kubernetes API的访问方式，最后详细阐述Kubernetes的命令行工具，帮助读者真正掌握如何使用Kubernetes。

8.1 API对象与元数据

Kubernetes中的很多功能是通过API对象来实现的，在前面的章节中我们已经创建过许多API对象，包括Pod、Replication Controller、Service和Secret等。在定义API对象的时候需要分别声明API版本（apiVersion）和类型（kind），当前版本Kubernetes v1.1.1支持的API对象的API版本和类型如下所示：

- v1/Pod
- v1/ReplicationController
- v1/Service
- v1/Endpoints

- v1/Events
- v1/Node
- v1/Namespace
- v1/Secret
- v1/ServiceAccount
- v1/PersistentVolume
- v1/PersistentVolumeClaim
- v1/LimitRange
- v1/ResourceQuota
- extensions/v1beta1/Deployment
- extensions/v1beta1/HorizontalPodAutoscaler
- extensions/v1beta1/Ingress
- extensions/v1beta1/Job
- extensions/v1beta1/Daemonset

API对象的元数据用来定义API对象的基本信息，体现在定义中的metadata字段，包含以下属性。

- namespace: 指定API对象所在的Namespace。

- **name:** 指定API对象的名称。
- **labels:** 设置API对象的Label。
- **annotations:** 设置API对象的Annotation。

Namespace

Namespace是Kubernetes提供的多租户，不同的项目、团队或者用户可以通过Namespace进行区分管理，并且设置安全控制和其他策略。绝大部分API对象（除了Node）归属于Namespace，API对象通过`.metadata.namespace`指定Namespace，如果没有指定Namespace，那么就是归属于默认Namespace `default`。

Name

名称是一个重要的属性，是人类可读的，元数据中的`.metadata.name`用于指定API对象的名称。Kubernetes系统中的API对象必须能够通过名称唯一标识，Kubernetes包含Namespace的逻辑层级，大部分API对象必须归属于Namespace，所以这些API对象的名称必须在Namespace内唯一。而另外对于Node和Namespace来说，需要在Kubernetes系统中唯一。

Label

Label用于区分API对象的Key/Value对，Label存放的应该是具有标识性的数据，Kubernetes通过Label可以对API对象进行选择。Replication Controller和Service都是通过Label关联Pod，而Pod也可以通过Label选择Node。

Annotation

Annotation用于存放用户的自定义数据，Annotation存放的是非标识的数据，所以不能像Label一样进行对象选择。但是Annotation的数据可以是长数据，可以有结构或者无结构，作为Label的一种补充，Annotation也是Key/Value对：

```
annotations:  
  key1: value1  
  key2: value2
```

8.2 如何访问Kubernetes API

使用Kubernetes都需要访问其API，而Kubernetes API Server作为Kubernetes系统的入口，以REST API接口方式提供给外部调用，所以访问Kubernetes API实际上就是调用Kubernetes API Server。其中Kubernetes API Server集成了Swagger，可以通过界面查询所有API的详细信息（<http://kube-master:8080/swagger-ui/>），如图8-1所示。

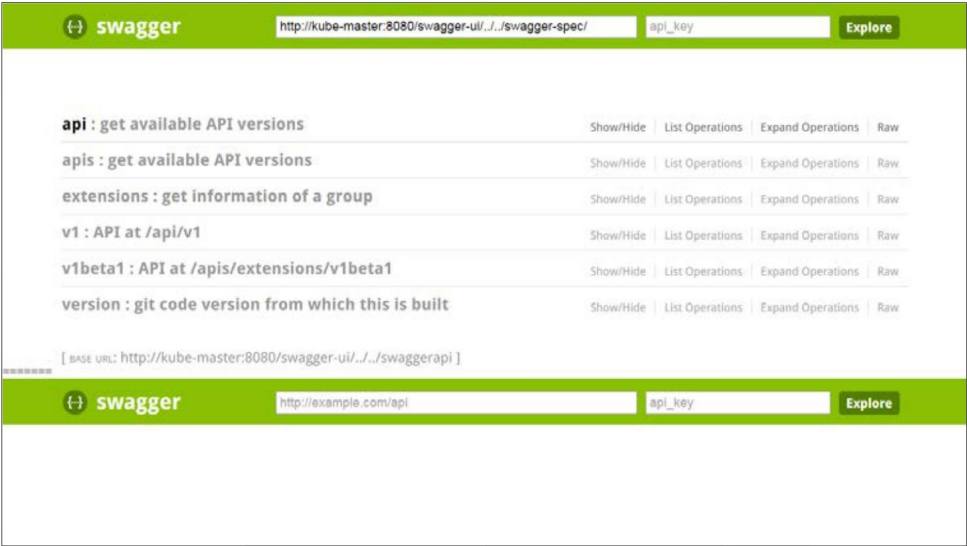


图8-1 Kubernetes集成Swagger提供API查询

除此之外，社区中封装提供了各种开发语言的Kubernetes客户端Lib包，如图8-1所示，可以使用这些Lib包来开发程序以访问Kubernetes API。

表8-1 Kubernetes客户端Lib包

Lib包	语言	URL
Kubernetes Client	Go	https://github.com/kubernetes/kubernetes/tree/v1.1.1/pkg/client
Amdatu Kubernetes	Java	https://bitbucket.org/amdatulabs/amdatu-kubernetes
kubernetes-client	Java	https://github.com/fabric8io/kubernetes-client
kubr	Ruby	https://github.com/Ch00k/kubr
kubeclient	Ruby	https://github.com/abonas/kubeclient
kubernetes-client	PHP	https://github.com/maclof/kubernetes-client
kubernetes-api-php-client	PHP	https://github.com/devstuf/kubernetes-api-php-client
node-kubernetes-client	Node.js	https://github.com/tenxcloud/node-kubernetes-client
Net::Kubernetes	Perl	https://github.com/kubernetes/kubernetes/blob/v1.1.1/docs/devel/client-libraries.md

8.3 使用命令行工具kubectl

Kubernetes提供了命令行工具kubectI，它提供了非常简洁快速的方法来访问Kubernetes API，可以满足大部分对Kubernetes的操作。kubectI可以从Kubernetes发布包中获取，其中platforms目录下放置着各个平台的kubectI可执行文件：

```
$                                wget
https://github.com/kubernetes/kubernetes/releases/download/v1.1
.1/kubernetes.tar.gz
$ tar zxvf kubernetes.tar.gz
$ cd kubernetes/platforms
```

命令行工具kubectI包含的命令如表8-2所示。

表8-2 kubectI命令

命令	说明
kubectI config	操作Kubeconfig文件
kubectI version	查询Kubernetes版本信息
kubectI api-versions	查询Kubernetes支持的API版本信息
kubectI cluster-info	查询Kubernetes运行环境信息
kubectI proxy	为Kubernetes API Server启动服务代理
kubectI create	创建API对象
kubectI delete	删除API对象
kubectI edit	编辑API对象
kubectI apply	更新API对象
kubectI patch	为API对象打补丁
kubectI label	操作API对象的Label
kubectI annotate	操作API对象的Annotation
kubectI logs	获取Pod中容器的输出日志
kubectI autoscale	执行Pod的自动伸缩
kubectI rolling-update	执行Pod的滚动升级
kubectI scale	执行Pod的弹性伸缩
kubectI attach	连接Pod中启动的容器
kubectI exec	在Pod的容器中执行命令
kubectI port-forward	为Pod设置端口转发
kubectI run	创建Replication Controller
kubectI expose	创建Service

8.3.1 配置Kubeconfig

使用kubectl命令行的时候首先需要配置Kubeconfig文件，用于配置如何访问Kubernetes API，包括Kubernetes API Server的URL和认证信息等，并且可以设置不同的上下文环境，快速切换访问环境。

下面是一个Kubeconfig文件示例：

```
apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority: /etc/kubernetes/ca/ca.crt
    server: https://kube-master:6443
  name: k8s
users:
- name: k8s-client
  user:
    client-certificate: /etc/kubernetes/ca/client.crt
    client-key: /etc/kubernetes/ca/client.key.insecure
- name: k8s-admin
  user:
    password: test
    username: k8s-admin
contexts:
- context:
    cluster: k8s
```

```
user: k8s-admin
namespace: default
name: default
current-context: default
preferences: {}
```

Kubeconfig文件的定义中包含以下关键部分。

- clusters: 设置Kubernetes API Server的访问URL和相关属性。
- users: 设置访问Kubernetes API Server的认证信息。
- contexts: 设置kubelet执行上下文。
- current-context: 设置kubelet执行当前上下文。
- preferences: 设置kubelet其他属性。

Kubeconfig文件可以手动进行编辑，也可以通过kubectl config命令进行查询和设置，如表8-3所示。

表8-3 kubectl config命令

命令	说明
kubectl config view	查看Kubeconfig文件
kubectl config set-cluster	设置Kubeconfig的clusters
kubectl config set-credentials	设置Kubeconfig的users
kubectl config set-context	设置Kubeconfig的contexts
kubectl config use-context	设置Kubeconfig的current-context

8.3.2 Kubernetes操作

kubectl version

`kubectl version`命令用来查询Kubernetes的版本信息，包括客户端和服务端，这在问题定位的时候特别有帮助：

\$ kubectl version

```
Client Version:
version.Info{Major:"1",Minor:"1",GitVersion:"v1.1.1",GitCommit:
"...",
GitTreeState:"clean"}
Server Version:
version.Info{Major:"1",Minor:"1",GitVersion:"v1.1.1",GitCommit:
"...",
GitTreeState:"clean"}
```

kubectl api-versions

`kubectl api-versions`命令可以查询Kubernetes支持的API版本：

\$ kubectl api-versions

```
extensions/v1beta1
v1
```

kubectl cluster-info

`kubectl cluster-info`命令可以查询Kubernetes的运行环境信息，包括Kubernetes API Server 和平台级别Service（kubernetes.io/cluster-service=true）的地址：


```
$ kubectl cluster-info
```

```
Kubernetes master is running at http://k8s-master:8080
```

```
KubeDNS          is          running          at          http://k8s-  
master:8080/api/v1/proxy/namespaces/kube-system/services/  
kube-dns
```

```
KubeUI           is          running          at          http://k8s-  
master:8080/api/v1/proxy/namespaces/kube-system/services/  
kube-ui
```

```
...
```

kubectl proxy

`kubectl proxy`命令可以为Kubernetes API Server在本地启动一个代理服务，访问这个代理服务就可以访问Kubernetes API Server。

执行`kubectl proxy`的时候可以指定监听端口和API访问前缀：

```
$ kubectl proxy --port=8011 --api-prefix=/k8s-api
```

```
Starting to serve on 127.0.0.1:8011
```

`kubectl proxy`启动后就可以通过`http://127.0.0.1:8011/k8s-api`访问到代理服务，从而访问Kubernetes API Server。

8.3.3 API对象操作

`kubectl`命令行可以操作API对象，执行的时候需要指定API对象的类型，类型可以用单数形式、复数形式或者简写形式，具体如下所

示:

- pod/pods/po
- replicationcontroller/replicationcontrollers/rc
- daemonset/daemonsets/ds
- service/services/svc
- endpoints/ep
- event/events/ev
- node/nodes/no
- namespace/namespaces/ns
- secret/secrets
- serviceaccount/serviceaccounts
- persistentvolume/persistentvolumes/pv
- persistentvolumeclaim/persistentvolumeclaims/pvc
- limitrange/limitranges/limits
- resourcequota/resourcequotas/quota
- componentstatuses/cs
- daemonset/daemonsets/ds

- deployment/deployments
- horizontalpodautoscaler/horizontalpodautoscalers/hpa
- ingress/ingresses/ing
- job/jobs

kubectl create

`kubectl create`命令用来创建API对象，格式支持JSON和YAML。使用`kubectl create`主要是通过定义文件进行创建：

```
$ kubectl create -f /path/to/file
```

也可通过传递标准输入（stdin）进行创建：

```
$ cat /path/to/file | kubectl create -f -
```

`kubectl create`支持连续创建多个API对象：

```
$ kubectl create -f /path/to/file1 -f /path/to/file2
```

kubectl get

`kubectl get`命令可以用来查询各类API对象的信息。

`kubectl get`可以查询指定API对象：

```
$ kubectl get TYPE NAME
```

可以查询一个类型的所有API对象：

```
$ kubectl get TYPE
```

也可以同时查询多个类型，类型之间用,分隔：

```
$ kubectl get TYPE1,TYPE2
```

kubectl get支持通过Label筛选API对象：

```
$ kubectl get TYPE --selector key1=value1,key2=value
```

默认情况下，kubectl get只会显示简要信息，以下方式可以显示详细信息：

```
$ kubectl get TYPE NAME --output json
```

```
$ kubectl get TYPE NAME --output yaml
```

kubectl get也支持通过Go Template或者JSON Path提取指定信息：

```
$ kubectl get TYPE NAME --output go-template=...
```

```
$ kubectl get TYPE NAME --output jsonpath=...
```

kubectl describe

kubectl describe命令可以用来查询各类API对象的概况信息，支持批量查询和指定查询：

```
$ kubectl describe TYPE
```

```
$ kubectl describe TYPE NAME
```

kubectl delete

kubectl delete命令用来删除API对象，删除的时候可以指定API对象进行删除：

```
$ kubectl delete TYPE NAME
```

也可以通过定义文件删除：

```
$ kubectl delete -f /path/to/file
```

另外，kubectl delete命令可以进行批量删除，多个API对象类型之间用,分隔：

```
$ kubectl delete TYPE1,TYPE2 --all
```

或者通过Label筛选删除：

```
$ kubectl delete TYPE1,TYPE2 --selector key1=value1,key2=value
```

kubectl apply

kubectl apply命令可以用来更新已创建的API对象，主要是通过定义文件进行修改：

```
$ kubectl apply -f /path/to/file
```

也可通过传递标准输入（stdin）进行修改：

```
$ cat /path/to/file | kubectl apply -f -
```

kubectl replace

kubectl replace命令与kubectl apply类似，都可以用来更新已创建的API对象：

```
$ kubectl apply -f /path/to/file
```

但有时候 API 对象的有些属性无法直接更新，这时候可以使用 kubectl replace命令强制进行重建以实现更新：

```
$ kubectl apply -f /path/to/file --force
```

kubectl edit

kubectl edit命令可以用来编辑已创建的API对象，使用kubectl edit命令的时候会开启一个编辑器。通过编辑器可以方便地对API对象进行编辑，默认的编辑器是VI，可以通过环境变量KUBE_EDITOR选择其他编辑器：

```
$ KUBE_EDITOR="nao" kubectl edit TYPE NAME
```

在编辑器中默认显示的格式是YAML，也可以指定为JSON：

```
$ kubectl edit TYPE NAME --output json
```

kubectl patch

kubectl patch命令可以给API对象打补丁，即修改指定属性，这样可以非常方便地修改API对象，其中PATCH要使用JSON格式：

```
$ kubectl patch TYPE --patch PATCH
```

kubectl label

kubectl label命令可以用来操作API对象的Label，包括增加、修改和删除。

给API对象增加Label，多个Label之间用空格分隔：

```
$ kubectl label TYPE NAME label1=value1 label2=value2
```

更新API对象已有的Label，需要加上--overwrite参数进行覆盖：

```
$ kubectl label TYPE NAME label1=new-value --overwrite
```

删除API对象已有的Label，在Label的KEY后面加上-：

```
$ kubectl label TYPE NAME label1-
```

另外，kubectl label也支持通过参数--all和--selector进行批量操作。

kubectl annotate

kubectl annotate命令可以用来操作API对象的Annotation，kubectl annotate同kubectl label命令差不多，只不过操作的对象换成Annotation。

给API对象增加Annotation：

```
$ kubectl annotate TYPE NAME annotation1=value1  
annotation2=value2
```

更新API对象已有的Annotation，需要加上--overwrite参数进行覆盖：

```
$ kubectl annotate TYPE NAME annotation1=new-value --overwrite
```

删除API对象已有的Annotation，在Annotation的KEY后面加上-:

```
$ kubectl annotate TYPE NAME annotation1-
```

8.3.4 Pod操作

kubectl logs

`kubectl logs`命令用于打印Pod中容器的日志输出，如果Pod只有一个容器，不需要指定容器:

```
$ kubectl logs mypod
```

而当Pod有多个容器的时候，需要指定容器:

```
$ kubectl logs mypod container
```

`kubectl logs`默认会打印所有日志，`--limit-bytes`参数可以限定日志打印量，而通过`--tail`参数可以只打印最新的指定行数的日志，或者通过`--since`和`--since-time`打印指定时间的日志。另外，通过设置`--follow`参数，将实时打印日志流，达到`tail -f`的效果。

kubectl attach

`kubectl attach`命令用于连接到Pod中启动的容器，类似于`docker attach`，如果不指定容器，则选择Pod的第一个容器:


```
$ kubectl attach mypod
```

也可以指定容器:

```
$ kubectl attch mypod container
```

kubectl exec

`kubectl exec` 命令用于在Pod的容器中执行命令，类似于 `docker exec`，如果不指定容器，则选择Pod的第一个容器:

```
$ kubectl exec mypod -- date
```

当然可以指定容器执行:

```
$ kubectl exec mypod container -- date
```

提示

`kubectl exec` 命令需要在Kubernetes Node上安装 `nsenter`。

kubectl port-forward

`kubectl port-forward` 命令可以为Pod设置端口转发，通过在本机监听指定端口，访问这些端口的请求将会被转发到Pod的容器中对应的端口上。

执行 `kubectl port-forward` 命令的时候需要指定 Pod 和端口转发规则，比如 80 端口转发 80 端口，443 端口转发 443 端口：

```
$ kubectl port-forward mypod 80:80 443:443
```

提示

`kubectl port-forward` 命令需要在 Kubernetes Node 上安装 `nsenter` 和 `socat`。

8.3.5 Replication Controller 操作

`kubectl run`

`kubectl run` 命令可以用来创建 Replication Controller，创建的时候必须指定容器镜像：

```
$ kubectl run nginx --image nginx
```

创建的 Replication Controller 的 Pod 副本数是 1，可以通过 `--replicas` 参数设置 Pod 的副本数为 2：

```
$ kubectl run nginx --image nginx --replicas 2
```

默认情况下，创建出来的 Replication Controller 会为 Pod 设置一个 Label，Label 的 Key 为 `run`，Value 为 Replication Controller 的名称。如果

kubectl run命令中设置了--labels参数，则会覆盖这个Label。

kubectl run命令中只可以设置一个容器，支持容器的属性设置如下所示。

- --command: 容器的启动命令。
- --port: 容器内部的端口。
- --hostport: 容器映射到宿主机的端口。
- --env: 容器的环境变量。
- --requests: 容器的资源请求规格。
- --limits: 容器的资源限制规格。

kubectl scale

kubectl scale命令可以用来修改Replication Controller的Pod副本数，即实现Pod的弹性伸缩。执行的时候需要指定Replication Controller和Pod副本数：

```
$ kubectl scale replicationcontroller nginx --replicas=3
```

kubectl scale命令如果设置了--current-replicas参数，那么会验证当前Pod的副本数是否等于--current-replicas配置的数目，相等才进行修改操作。

kubectl autoscale

kubectl autoscale 命令可以为 Replication Controller 创建 Horizontal Pod Autoscaler，即实现 Pod 的自动伸缩。执行的时候需要指定 Replication Controller，以及 Pod 的最大和最小副本数：

```
$ kubectl autoscale replicationcontroller nginx --min=1 --max=10
```

8.3.6 Service操作

kubectl expose

kubectl expose 命令可以用来创建 Service，创建的时候需要指定 Pod、Replication Controller 或者 Service，从中提取 Label 来为新建的 Service 配置 Label Selector：

```
$ kubectl expose pod valid-pod --port=444 --name=frontend
$ kubectl expose replicationcontroller nginx --port=80 --target-port=8000
$ kubectl expose service nginx --port=443 --target-port=8443 --name=nginx-https
```

第2部分

Kubernetes高级篇

第9章 Kubernetes网络

第10章 Kubernetes安全

第11章 Kubernetes资源管理

第12章 管理和运维Kubernetes

第9章

Kubernetes网络

Kubernetes从Docker默认的网络模型中独立出来形成一套自己的网络模型，该网络模型更加适应传统的网络模式，应用能够平滑地从非容器环境迁移到同Kubernetes中。本章将对比说明Kubernetes和Docker的网络模型，然后详细讲解Kubernetes网络模型的实现细节。

9.1 Docker网络模型

Docker 使用 Linux 桥接，在宿主机上虚拟一个 Docker 网桥（docker0），Docker启动一个容器时会根据Docker网桥的网段分配容器的IP，同时Docker网桥是每个容器的默认网关。因为在同一宿主机内的容器都接入同一个网桥，这样容器之间就能通过容器的IP直接通信，如图9-1所示。

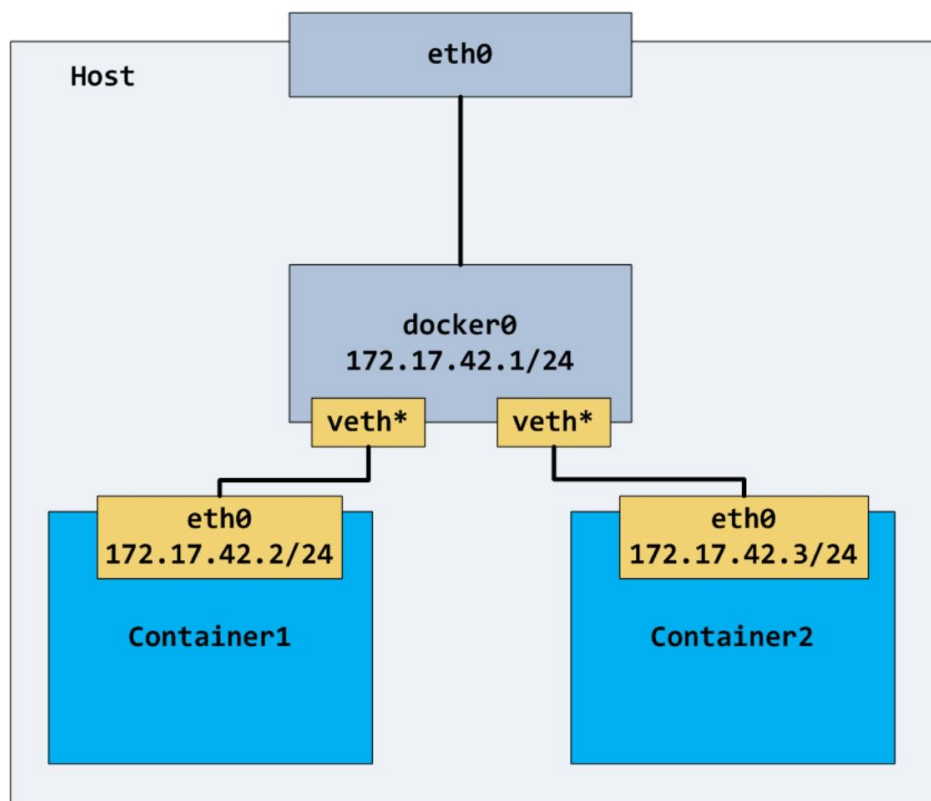


图9-1 Docker网络

Docker网桥是宿主机虚拟出来的，并不是真实存在的网络设备，外部网络是无法寻址到的，这也意味着外部网络无法直接访问到容器。如果容器希望能够被外部网络访问到，就需要通过映射容器端口到宿主机（端口映射），即使用docker run创建容器时通过-p或-P参数来启用，访问容器的时候通过[宿主机IP]:[容器端口]访问容器。

实际上，端口映射通过在iptables的NAT表中添加相应的规则，所以我们也称端口映射方式称为NAT方式。在早期组建Docker机器集群的方案中，往往是选择了NAT方式的网络模型。这种网络模型对使用的便利性是有意义的，但并不理想。这个模型需要对各种端口进行映射，这会限制宿主机的能力，在容器编排上也增加了复杂度。

- 端口是个稀缺资源，这就需要解决端口冲突和动态分配端口问题。这不但使调度复杂化，而且应用程序的配置也将变得复杂，具体表现为端口冲突、重用和耗尽。

- NAT将地址空间分段的做法引入了额外的复杂度。比如容器中应用所见的IP并不是对外暴露的IP，因为网络隔离，容器中的应用实际上只能检测到容器的IP，但是需要对外宣称的则是宿主机的IP，这种信息的不对称将带来诸如破坏自注册机制等问题。

9.2 Kubernetes网络模型

Kubernetes从Docker网络模型（NAT方式的网络模型）中独立出来形成一套新的网络模型。该网络模型的目标是：每一个Pod都拥有一个扁平化共享网络命名空间的IP，称为PodIP。通过PodIP，Pod能够跨网络与其他物理机和Pod进行通信。一个Pod一个IP的（IP-Per-Pod）模型创建了一个干净、反向兼容的模型。在该模型中，从端口分配、网络、域名解析、服务发现、负载均衡、应用配置和迁移等角度，Pod都能够被看成虚拟机或物理机，这样应用就能够平滑地从非容器环境（物理机或虚拟机）迁移到同一个Pod内的容器环境。

为了实现这个网络模型，在Kubernetes中需要解决几个问题：

- 容器间通信（Container to Container）
- Pod间通信（Pod to Pod）
- Service到Pod的通信（Service to Pod）

9.3 容器间通信

Pod是容器的集合，Pod包含的容器都运行在同一个宿主机上，这些容器将拥有同样的网络空间，容器之间能够互相通信，它们能够在本地访问其他容器的端口。

现在创建一个Web Pod，包含两个容器：容器webpod80将启动监听80端口的Web服务，容器webpod8080将启动监听8080端口的Web服务，同时配置了端口映射规则。Web Pod的定义文件web-pod.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: webpod
  labels:
    name: webpod
spec:
  containers:
    - name: webpod80
      image: jonlangemak/docker:web_container_80
      ports:
        - containerPort: 80
          hostPort: 80
    - name: webpod8080
      image: jonlangemak/docker:web_container_8080
      ports:
```

```
- containerPort: 8080
  hostPort: 8080
```

Web Pod运行成功后，在其所在的Node上查询容器：

```
$ docker ps
CONTAINER ID   IMAGE                                PORTS
63dc7e032ab6   jonlangemak/docker:web_container_8080
4ac1a5156a04   jonlangemak/docker:web_container_80
b77896498f8f   gcr.io/google_containers/pause:0.8.0
0.0.0.0:80->80/tcp,0.0.0.0:8080->8080/tcp
```

可以看到运行了3个容器，其中前两个容器是在Web Pod定义中指定的，可以称为业务容器。第3个运行的容器镜像是gcr.io/google_containers/pause:0.8.0，Web Pod定义中设置的业务容器的端口映射规则都集中配置在了网络容器上，实际上它是Kubernetes中定义的网络容器，它不做任何事情，只是用来接管Pod的网络，业务容器通过加入网络容器的网络实现网络共享。

那么为什么要使用额外的网络容器，而不是以Pod的第一个容器作为网络容器呢？主要是为了避免业务容器之间产生依赖，比如第二个容器连接到第一个容器，当第一个容器宕机时，那么第二个容器的网络栈也会失效。

网络容器镜像可以通过Kubelet的启动参数--pod-infra-container-image指定，不同版本下使用的默认镜像不一样，当前版本（Kubernetes v1.1.1）默认是gcr.io/google_containers/pause:0.8.0，这是

一个非常小的镜像，启动容器后只会运行一个叫作`pause`的程序，顾名思义，`pause`的作用只是暂停，防止容器退出。

实际上，Kubernetes利用了Docker的容器网络共享能力，Web Pod中的容器类似于使用以下命令运行，而Pod的PodIP便是网络容器的IP：

```
$ docker run -p 80:80 -p 8080:8080 --name network-container -d
gcr.io/google_containers/pause:0.8.0
$ docker run --net container:network-container -d
jonlangemak/docker:web_container_80
$ docker run --net container:network-container -d
jonlangemak/docker:web_container_8080
```

这样一来，Pod中的所有容器都是互通的，而Pod对外可以看成是一个完整网络单元，如图9-2所示。

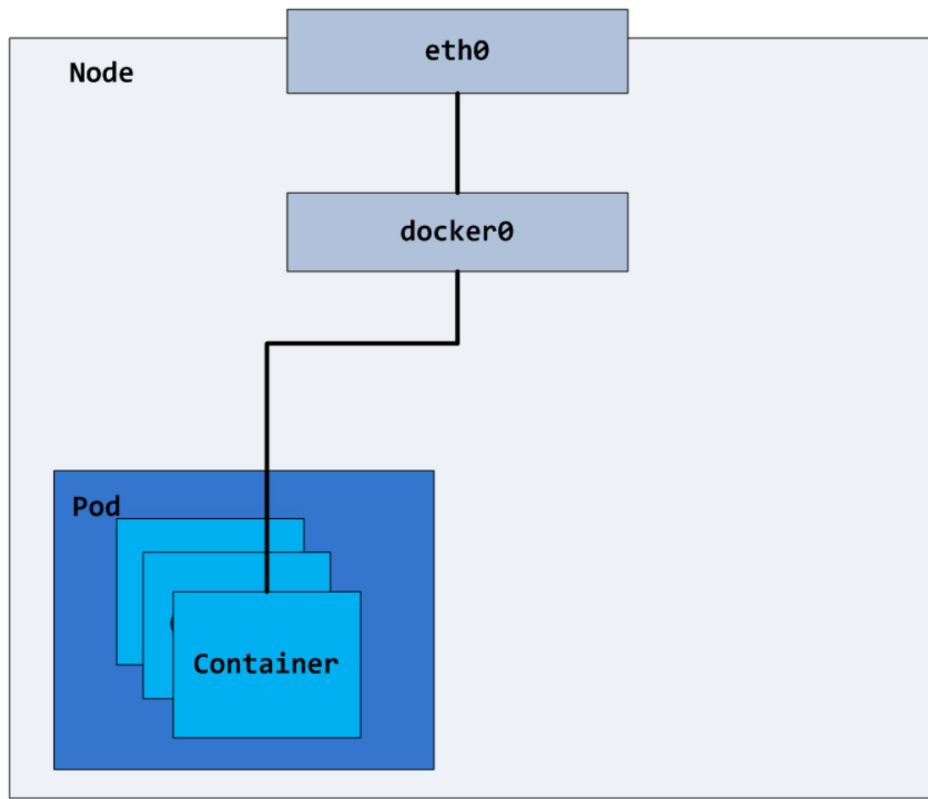


图9-2 Pod中的容器网络

9.4 Pod间通信

Kubernetes网络模型是一个扁平化的网络平面，在这个网络平面内，Pod作为一个网络单元同Kubernetes Node的网络处于同一层级。

我们考虑一个最小的Kubernetes网络拓扑，如图9-3所示，在这个网络拓扑中满足以下条件。

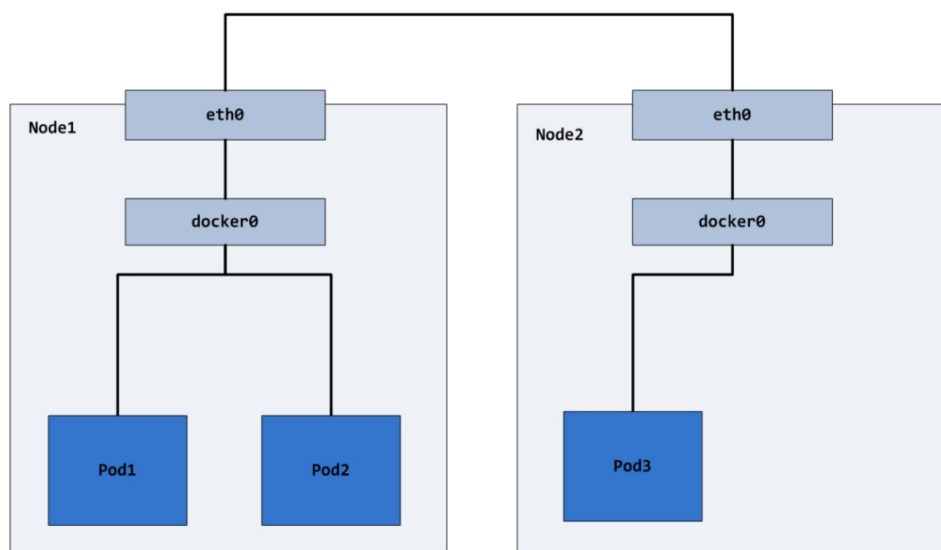


图9-3 Kubernetes网络拓扑

- Pod间通信：Pod1和Pod2（同主机），Pod1和Pod3（跨主机）能够通信。

- Node与Pod间通信：Node1和Pod1/ Pod2（同主机），Pod3（跨主机）能够通信。

那么第一个问题是如何保证Pod的PodIP是全局唯一的。其实做法也很简单，因为Pod的PodIP是Docker网桥分配的，所以将不同Kubernetes Node的Docker网桥配置成不同的IP网段即可。

另外，同一个Kubernetes Node上的Pod/容器原生能通信，但是Kubernetes Node之间的Pod/容器是如何通信的，这就需要对Docker进行增强，在容器集群中创建一个覆盖网络（Overlay Network），联通各个节点，目前可以通过第三方网络插件来创建覆盖网络，比如Flannel和Open vSwitch等。

9.4.1 Flannel实现Kubernetes覆盖网络

Flannel是由CoreOS团队设计开发的一个覆盖网络工具，它通过在集群中创建一个覆盖网络，为主机设定一个子网，通过隧道协议封装容器之间的通信报文，实现容器的跨主机通信。

现在我们利用Flannel联通两个Kubernetes Node，如表9-1所示。

表9-1 Kubernetes Node信息

节点	主机名	IP
Kubernetes Node 1	kube-node-1	192.168.3.147
Kubernetes Node 2	kube-node-2	192.168.3.148

Flannel使用Etcd作为配置和协调中心，在运行Flannel之前需要在Etcd中进行配置。Flannel的Etcd配置目录可以通过启动参数-etcd-prefix指定，默认是/coreos.com/network。

首先需要在Etcd上配置Flannel网络信息：

```
$ etcdctl set /coreos.com/network/config '{ "Network":  
"10.0.0.0/16" }'
```

配置完成后，在Kubernetes Node上运行Flannel，Flannel在初次启动的时候会检查Etcd中的配置，然后为当前节点分配可用的IP网段，然后在Etcd中创建一个路由表，通过Etcd查询路由表：

```
$ etcdctl ls /coreos.com/network/subnets
```

```
/coreos.com/network/subnets/10.0.10.0-24
```

```
/coreos.com/network/subnets/10.0.62.0-24
```

```
$ etcdctl get /coreos.com/network/subnets/10.0.62.0-24
```

```
{"PublicIP":"192.168.3.147"}
```

```
$ etcdctl get /coreos.com/network/subnets/10.0.10.0-24
```

```
{"PublicIP":"192.168.3.148"}
```

Flannel 在分配到 IP 网段之后，会创建一个虚拟网卡，在 Kubernetes Node 1 上查询Flannel虚拟网卡：

```
$ ip addr show flannel.1
```

```
5: flannel.1: <BROADCAST, MULTICAST, UP, LOWER_UP> mtu 1450
```

```
qdisc noqueue state UNKNOWN
```

```
    link/ether 62:71:56:28:bd:dd brd ff:ff:ff:ff:ff:ff
```

```
    inet 10.0.62.0/16 scope global flannel.1
```

```
        valid_lft forever preferred_lft forever
```

```
    inet6 fe80::6071:56ff:fe28:bddd/64 scope link
```

```
        valid_lft forever preferred_lft forever
```

另外，Flannel会配置Docker网桥（docker0），实际上就是通过修改Docker的启动参数--bip来实现。这样一来，集群中每个节点的Docker网桥就分配好了全局唯一的IP网段，从而创建出来的容器也将拥有全局唯一的IP，比如在Kubernetes Node 1上查询Docker网桥：


```
$ ip addr show docker0
6: docker0: <NO-CARRIER, BROADCAST, MULTICAST, UP> mtu 1450
qdisc noqueue state DOWN
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 10.0.62.1/24 scope global docker0
        valid_lft forever preferred_lft forever
    inet6 fe80::5484:7aff:fefe:9799/64 scope link
        valid_lft forever preferred_lft forever
```

由此，Flannel为两个Kubernetes Node划分好了容器网络网段，如表9-2所示。

表9-2 Kubernetes Node容器网络网段划分（Flannel）

节点	主机名	IP	Docker网桥
Kubernetes Node 1	kube-node-1	192.168.3.147	10.0.62.1/24
Kubernetes Node 2	kube-node-2	192.168.3.148	10.0.10.1/24

除此之外，Flannel会修改路由表，使得Flannel虚拟网卡可以接管容器跨主机的通信。在Kubernetes Node 1上查询路由表：

```
$ route -n
Kernel IP routing table
Destination      Gateway          Genmask          Flags  Metric
```

Ref	Use	Iface			
10.0.0.0		0.0.0.0	255.255.0.0	U	0
0	0	flannel.1			
10.0.62.0		0.0.0.0	255.255.255.0	U	0
0	0	docker0			
...					

在Kubernetes Node 2上查询路由表:

```
$ route -n
```

Kernel IP routing table

Destination	Gateway	Genmask	Flags	Metric
Ref	Use	Iface		
10.0.0.0		0.0.0.0	255.255.0.0	U 0
0	0	flannel.1		
10.0.10.0		0.0.0.0	255.255.255.0	U 0
0	0	docker0		
...				

这样一来，当一个节点的容器访问另一个节点的容器时，源节点上的数据会从docker0网桥路由到flannel1.1网卡，在目的节点会从flannel 1.1网卡路由到docker0网桥。比如现在有一个数据包要从IP为10.0.62.2的容器发到IP为10.0.10.2的容器，根据Kubernetes Node 1的路由表，它只与10.0.0.0/16这条记录匹配，因此数据包从docker0网桥出来以后就被路由到了flannel1.1网卡。同理，在Kubernetes Node 2上，由于目的地址匹配在docker0网桥对于的10.0.10.0/24这个记录上，数据包从flannel1.1网卡出来以后就被路由到了docker0网桥，最后转发给目标容器。

Flannel虚拟网卡接收到的数据包会被Flannel服务进行封装，Flannel将通过隧道协议封装这些数据包，目前隧道协议已经支持UDP、VxLAN等，默认是UDP。当容器跨主机通信的时候，源主机的Flannel服务将接收到的数据包包装在另一种网络包中，然后目的主机的Flannel服务再进行解包。

最终，Flannel将运行在所有Kubernetes Node上，Flannel重新规划容器集群网络，从而使得集群中所有容器能够获得同属一个内网且不重复的IP，并让属于不同节点上的容器能够直接通过内网IP通信，Flannel实现的网络结构如图9-4所示。

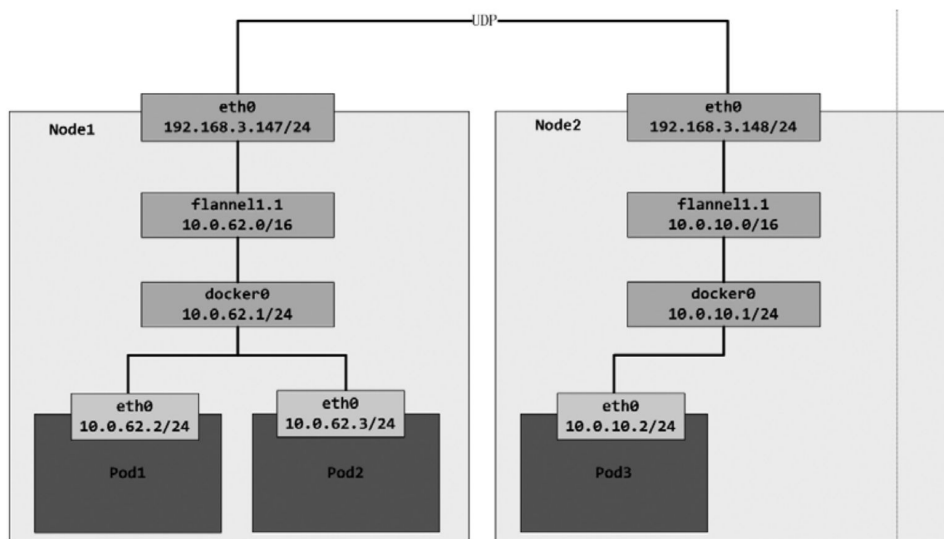


图9-4 Flannel实现Kubernetes覆盖网络

9.4.2 使用Open vSwitch实现Kubernetes覆盖网络

Open vSwitch是一个高质量的、多层虚拟交换机，使用开源Apache 2.0许可协议，由Nicira Networks开发。它的目的是让大规模网络自动化可以通过编程扩展，同时仍然支持标准的管理接口和协议。

Open vSwitch也提供了对OpenFlow协议的支持，用户可以使用任何支持OpenFlow协议的控制器对Open vSwitch进行远程管理控制。Open vSwitch是一项非常重要的SDN技术，可以灵活地创建出满足各种需求的虚拟网络，也包括Kubernetes中的覆盖网络。

现在我们利用Open vSwitch联通两个Kubernetes Node。为了保证容器IP不冲突，所以必须规划好Kubernetes Node上Docker网桥的网段，如表9-3所示。

表9-3 Kubernetes Node容器网络网段划分（Open vSwitch）

节点	主机名	IP	Docker网桥
Kubernetes Node 1	kube-node-1	192.168.3.147	10.246.0.1/24
Kubernetes Node 2	kube-node-2	192.168.3.148	10.246.1.1/24

Open vSwitch实现的网络模型如图9-5所示。

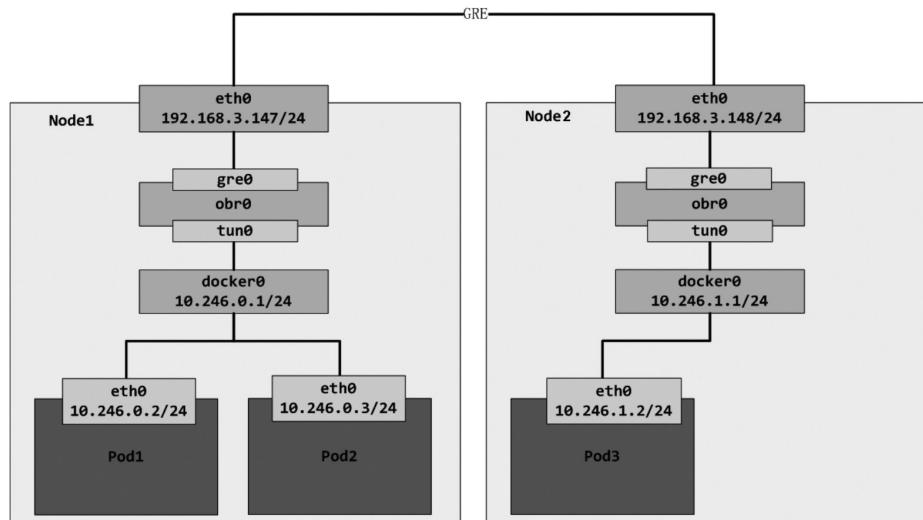


图9-5 Open vSwitch实现Kubernetes覆盖网络

1. 首先设置Docker网桥:

• Kubernetes Node 1

```
$ brctl addbr docker0
$ ip link set dev docker0 up
$ ifconfig docker0 10.246.0.1 netmask 255.255.255.0 up
```

• Kubernetes Node 2

```
$ brctl addbr docker0
$ ip link set dev docker0 up
$ ifconfig docker0 10.246.1.1 netmask 255.255.255.0 up
```

2. 然后创建Open vSwitch虚拟网桥obr0，连接各个网络设备:

• Kubernetes Node 1

#创建obr0

```
$ ovs-vsctl add-br obr0 -- set Bridge obr0 fail-mode=secure
```

```
$ ovs-vsctl set bridge obr0 protocols=OpenFlow13
```

#创建GRE隧道

```
$ ovs-vsctl add-port obr0 gre0 \
```

```
-- set Interface gre0 type=gre options:remote_ip=flow  
options:key=flow ofport_request=10
```

#创建tun0连接docker0和obr0

```
$ ovs-vsctl add-port obr0 tun0 -- set Interface tun0  
type=internal ofport_request=9
```

```
$ brctl addif docker0 tun0
```

```
$ ip link set tun0 up
```

#设置OpenFlow规则

```
$ ovs-ofctl -O OpenFlow13 del-flows obr0
```

```
$ ovs-ofctl -O OpenFlow13 add-flow obr0 \  
table=0,ip,in_port=10,nw_dst=10.246.0.1/24,actions=output:9
```

```
$ ovs-ofctl -O OpenFlow13 add-flow obr0 \  
table=0,arp,in_port=10,nw_dst=10.246.0.1/24,actions=output:9
```

```
$ ovs-ofctl -O OpenFlow13 add-flow obr0 \  
table=0,in_port=9,ip,nw_dst=10.246.1.1/24,actions=set_field:192  
.168.3.149->tun_dst,output:10
```

```
$ ovs-ofctl -O OpenFlow13 add-flow obr0 \  
table=0,in_port=9,arp,nw_dst=10.246.1.1/24,actions=set_field:19  
2.168.3.149->tun_dst, output:10
```

- Kubernetes Node 2

#创建obr0

```
$ ovs-vsctl add-br obr0 -- set Bridge obr0 fail-mode=secure
```

```
$ ovs-vsctl set bridge obr0 protocols=OpenFlow13
```

#创建GRE隧道

```
$ ovs-vsctl add-port obr0 gre0 \
```

```
-- set Interface gre0 type=gre options:remote_ip=flow  
options:key=flow ofport_request=10
```

#创建tun0连接docker0和obr0

```
$ ovs-vsctl add-port obr0 tun0 -- set Interface tun0  
type=internal ofport_request=9
```

```
$ brctl addif docker0 tun0
```

```
$ ip link set tun0 up
```

#设置OpenFlow规则

```
$ ovs-ofctl -O OpenFlow13 del-flows obr0
```

```
$ ovs-ofctl -O OpenFlow13 add-flow obr0 \  
table=0,ip,in_port=10,nw_dst=10.246.1.1/24,actions=output:9
```

```
$ ovs-ofctl -O OpenFlow13 add-flow obr0 \  
table=0,arp,in_port=10,nw_dst=10.246.1.1/24,actions=output:9
```

```
$ ovs-ofctl -O OpenFlow13 add-flow obr0 \  
table=0,in_port=9,ip,nw_dst=10.246.0.1/24,actions=set_field:192  
.168.3.148->tun_dst,output:10
```

```
$ ovs-ofctl -O OpenFlow13 add-flow obr0 \
```

```
table=0,in_port=9,arp,nw_dst=10.246.0.1/24,actions=set_field:19
2.168.3.148->tun_dst,
output:10
```

3. 最后配置路由:

- Kubernetes Node 1

```
$ ip route add 10.246.0.0/16 dev docker0 scope link src
10.246.0.1
```

- Kubernetes Node 2

```
$ ip route add 10.246.0.0/16 dev docker0 scope link src
10.246.1.1
```

9.5 Service到Pod通信

Service在Pod之间起到服务代理的作用，对外表现为一个单一访问接口，将请求转发给Pod，Service的网络转发是Kubernetes实现服务编排的关键一环。

现在有一个Service，查询详情如下：

```
$ kubectl describe service myservice
```

```
Name:    myservice
```

```
Namespace: default
```

```
Labels:   <none>
```



```
Selector: name=mypod
Type: ClusterIP
IP: 10.254.206.148
Port: http 80/TCP
Endpoints: 10.0.62.87:80,10.0.62.88:80,10.0.62.89:80
Session Affinity: None
No events.
```

可以看到，该Service的虚拟IP是10.254.206.148，端口80/TCP对应的 Endpoints 包含 3 个 后 端：10.0.62.87:80、10.0.62.88:80 和 10.0.62.89:80，即当请求10.254.149.17:80时，会转发到这些后端之一。

首先虚拟 IP 是由 Kubernetes 创建的，虚拟 IP 的网段是通过 Kubernetes API Server的启动参数--service-cluster-ip-range=10.254.0.0/16 配置的。

另一个关键组件是Kubernetes Proxy，Kubernetes Proxy组件负责实现虚拟IP路由和转发，而在容器覆盖网络之上又实现了Kubernetes层级的虚拟转发网络。Kubernetes Proxy需要满足以下功能：

- 转发访问Service的虚拟IP的请求到Endpoints。
- 监控Service和Endpoints的变化，实时刷新转发规则。
- 提供负载均衡能力。

在当前版本（Kubernetes v1.1.1）中，Kubernetes Proxy有两种实现机制：Userspace和Iptables，可以通过Kubernetes Proxy的启动参数--

proxy-mode指定。

9.5.1 Userspace模式

在Userspace模式下，Kubernetes Proxy将会为每一个Service在主机上启用随机端口进行监听，并且创建Iptables规则重定向访问Service虚拟IP的请求到这个端口上，而Kubernetes Proxy将请求转发到Endpoints。在此模式下，Kubernetes Proxy起到反向代理的作用，请求的转发由Kubernetes Proxy在用户空间下完成。Kubernetes Proxy需要监控Endpoints的变化，实时刷新转发规则，如图9-6所示。

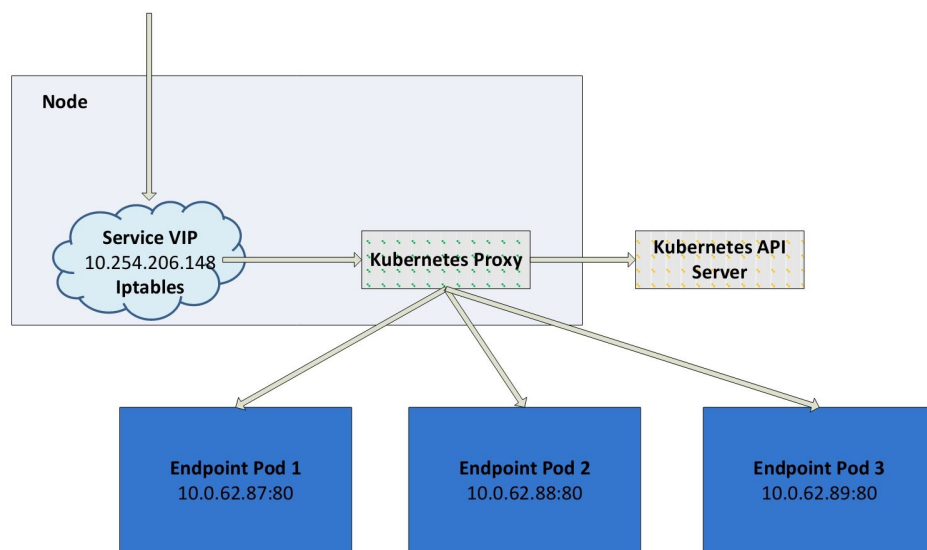


图9-6 Kubernetes Proxy的Userspace模式

当前Kubernetes Proxy只是3层（TCP/UDP Over IP）转发，默认的负载均衡策略是轮询方式。

通过iptables-save命令可查询关于Service myservice的Iptables规则：

```
$ iptables-save|grep myservice
```

```
-A KUBE-PORTALS-CONTAINER -d 10.254.206.148/32 -p tcp -m  
comment --comment  
"default/myservice:http" -m tcp --dport 80 -j REDIRECT --to-  
ports 35841  
-A KUBE-PORTALS-HOST -d 10.254.206.148/32 -p tcp -m comment --  
comment  
"default/myservice:http" -m tcp --dport 80 -j DNAT --to-  
destination 192.168.3.146:35841
```

Kubernetes Proxy会为Service创建两条Iptables规则，其中包含如下两个Iptables自定义链。

- KUBE-PORTALS-CONTAINER：用于匹配容器发出的报文，绑定在NAT表PREROUTING链。

- KUBE-PORTALS-HOST：用于匹配宿主机发出的报文，绑定在NAT表OUTPUT链。

对于Service myservice，两条Iptables规则的作用都是为了将目的IP为10.254.206.148/32，并且目的端口为80的报文重定向到本机的35841端口，而35841端口就是Kubernetes Proxy监听的端口，Kubernetes Proxy监听在35841，作为反向代理将请求转发到Service myservice的Endpoints。

```
$ lsof -i:35841
```

COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE/OFF	NODE
NAME							

```
kube-proxy 1040 root      5u    IPv6  20457      0t0      TCP
*:51056 (LISTEN)
```

9.5.2 Iptables模式

在Iptables模式下，Kubernetes Proxy则是完全通过创建Iptables规则，直接重定向访问Service虚拟IP的请求到Endpoints。而当Endpoints发生变化的时候，Kubernetes Proxy会刷新相关的Iptables规则。在此模式下，Kubernetes Proxy只是负责监控Service和Endpoints，更新Iptables规则，报文的转发依赖于Linux内核，默认的负载均衡策略是随机方式，如图9-7所示。

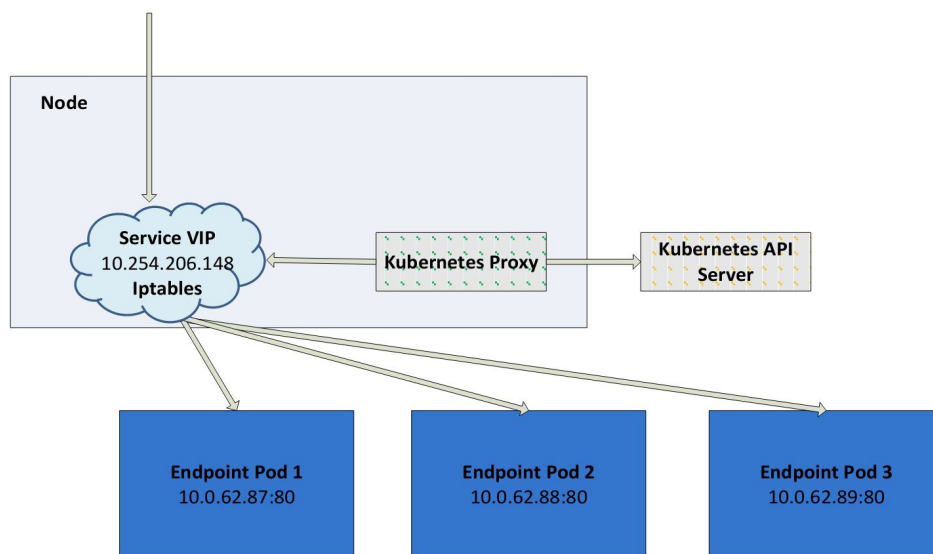


图9-7 Kubernetes Proxy的Iptables模式

通过iptables-save命令可查询到关于Service myservice的Iptables规则：

\$ iptables-save|grep myservice

```
-A KUBE-SEP-5534AF2Y644GLPZI -s 10.0.62.87/32 -m comment --
comment "default/myservice:
http" -j MARK --set-xmark 0x4d415351/0xffffffff
-A KUBE-SEP-5534AF2Y644GLPZI -p tcp -m comment --comment
"default/myservice:http" -m tcp
-j DNAT --to-destination 10.0.62.87:80
-A KUBE-SEP-5UWZC74U4HN6VNI5 -s 10.0.62.88/32 -m comment --
comment "default/myservice:
http" -j MARK --set-xmark 0x4d415351/0xffffffff
-A KUBE-SEP-5UWZC74U4HN6VNI5 -p tcp -m comment --comment
"default/myservice:http" -m tcp
-j DNAT --to-destination 10.0.62.88:80
-A KUBE-SEP-W2TMLVNCN2RGF2MT -s 10.0.62.89/32 -m comment --
comment "default/myservice:
http" -j MARK --set-xmark 0x4d415351/0xffffffff
-A KUBE-SEP-W2TMLVNCN2RGF2MT -p tcp -m comment --comment
"default/myservice:http" -m tcp
-j DNAT --to-destination 10.0.62.89:80
-A KUBE-SERVICES -d 10.254.206.148/32 -p tcp -m comment --
comment "default/myservice:http
cluster IP" -m tcp --dport 80 -j MARK --set-xmark
0x4d415351/0xffffffff
-A KUBE-SERVICES -d 10.254.206.148/32 -p tcp -m comment --
comment "default/myservice:http
cluster IP" -m tcp --dport 80 -j KUBE-SVC-YZPZUSJVGCLDKDHP
-A KUBE-SVC-YZPZUSJVGCLDKDHP -m comment --comment
```

```

"default/myservice:http" -m statistic
--mode random --probability 0.33332999982 -j KUBE-SEP-
5534AF2Y644GLPZI
-A KUBE-SVC-YZPZUSJVGCLDKDHP -m comment --comment
"default/myservice:http" -m statistic
--mode random --probability 0.500000000000 -j KUBE-SEP-
5UWZC74U4HN6VNI5
-A KUBE-SVC-YZPZUSJVGCLDKDHP -m comment --comment
"default/myservice:http" -j
KUBE-SEP-W2TMLVNCN2RGF2MT

```

Kubernetes Proxy会为Service创建一系列Iptables规则，其中包含Iptables自定义链。

- KUBE-SERVICES：绑定在NAT表PREROUTING链和OUTPUT链。
- KUBE-SVC-*：代表一个Service，绑定在KUBE-SERVICES。
- KUBE-SEP-*：代表Endpoints的每一个后端，绑定在KUBE-SVC-*。

对于Service myservice，Service对应的Iptables自定义链是KUBE-SVC-YZPZUSJVGCLDKDHP，Endpoints对应的Iptables自定义链是KUBE-SEP-5534AF2Y644GLPZI、KUBE-SEP-5UWZC74U4HN6VNI5和KUBE-SEP-W2TMLVNCN2RGF2MT。

通过iptables查询可以更加清晰地看出转发的规则：

```
$ iptables -t nat -L -n
```

```
Chain PREROUTING (policy ACCEPT)
```

target	prot	opt	source	destination
KUBE-SERVICES	all	--	0.0.0.0/0	0.0.0.0/0 /*
kubernetes service portals */				

```
Chain INPUT (policy ACCEPT)
```

target	prot	opt	source	destination
--------	------	-----	--------	-------------

```
Chain OUTPUT (policy ACCEPT)
```

target	prot	opt	source	destination
KUBE-SERVICES	all	--	0.0.0.0/0	0.0.0.0/0 /*
kubernetes service portals */				

```
Chain POSTROUTING (policy ACCEPT)
```

target	prot	opt	source	destination
MASQUERADE	all	--	0.0.0.0/0	0.0.0.0/0 /*
kubernetes service traffic				
requiring SNAT */ mark match 0x4d415351				

```
Chain KUBE-SERVICES (2 references)
```

target	prot	opt	source	destination
MARK	tcp	--	0.0.0.0/0	10.254.206.148 /*
default/myservice:http				
cluster IP */ tcp dpt:80 MARK set 0x4d415351				
KUBE-SVC-YZPZUSJVGCLDKDHP	tcp	--		0.0.0.0/0
10.254.206.148				/* default/
myservice:http cluster IP */ tcp dpt:80				

Chain KUBE-SVC-YZPZUSJVGCLDKDHP (1 references)

target	prot	opt	source	destination
KUBE-SEP-5534AF2Y644GLPZI	all	--		0.0.0.0/0
0.0.0.0/0			/* default/	
myservice:http	*/	statistic	mode	random probability
0.33332999982				
KUBE-SEP-5UWZC74U4HN6VNI5	all	--		0.0.0.0/0
0.0.0.0/0			/* default/	
myservice:http	*/	statistic	mode	random probability
0.500000000000				
KUBE-SEP-W2TMLVNCN2RGF2MT	all	--		0.0.0.0/0
0.0.0.0/0			/* default/	
myservice:http	*/			

Chain KUBE-SEP-5534AF2Y644GLPZI (1 references)

target	prot	opt	source	destination
MARK	all	--	10.0.62.87	0.0.0.0/0
/* default/			myservice:http	*/
MARK set			0x4d415351	
DNAT	tcp	--	0.0.0.0/0	0.0.0.0/0
/* default/			myservice:http	*/
tcp to:			10.0.62.87:80	

Chain KUBE-SEP-5UWZC74U4HN6VNI5 (1 references)

target	prot	opt	source	destination
MARK	all	--	10.0.62.88	0.0.0.0/0 /*


```
default/myservice:http */
```

```
MARK set 0x4d415351
```

```
DNAT          tcp    --    0.0.0.0/0          0.0.0.0/0
```

```
/* default/myservice:http */
```

```
tcp to:10.0.62.88:80
```

```
Chain KUBE-SEP-W2TMLVNCN2RGF2MT (1 references)
```

```
target      prot opt source                destination
```

```
MARK        all    --    10.0.62.89          0.0.0.0/0
```

```
/* default/myservice:http */
```

```
MARK set 0x4d415351
```

```
DNAT          tcp    --    0.0.0.0/0          0.0.0.0/0
```

```
/* default/myservice:http */
```

```
tcp to:10.0.62.89:80
```

第10章

Kubernetes安全

安全永远是一个重大的话题，特别是对于Kubernetes这样的云计算平台，更需要设计出一套完善的安全方案，以应对复杂的场景。本章将阐述Kubernetes的安全保障机制，主要包括Kubernetes API的安全访问、容器安全 and 多租户。

10.1 Kubernetes安全原则

Kubernetes设计出了一套API和敏感信息处理方案，也提供了容器安全保障，以下是Kubernetes的安全设计原则：

- 保证容器与其运行的宿主机之间有明确的隔离。
- 限制容器对基础设施或者其他容器造成不良影响。
- 最小特权原则——限定每个组件只被赋予执行操作所必需的最小特权，由此确保可能产生的损失最小。
- 普通用户明确区别于系统管理员。
- 能够给普通用户赋予管理权限。
- 应用能够安全地获取敏感信息。

10.2 Kubernetes API的安全访问

Kubernetes API Server是Kubernetes系统的入口，以REST API接口方式提供给外部客户和内部组件调用，对于Kubernetes API的访问调用需要进行严格管控，保证系统的安全。

10.2.1 HTTPS

Kubernetes API Server采用的是REST模式的API服务，REST利用传统Web特点，提出了一个既适于客户端应用又适于服务端应用的统一架构，极大程度上统一及简化了网站架构设计。

REST的全称是Representational State Transfer，表示表述性状态传递，无须Session，为了安全，每次请求都得带上身份认证信息。REST是基于HTTP的，也是无状态的，HTTP是明文传输的，所以建议所有的请求都通过HTTPS发送，保证对于系统中的重要数据做SSL加密传输，如证书、账号密码等。

Kubernetes API Server支持HTTP和HTTPS，相关启动参数如表10-1所示。

表10-1 Kubernetes API Server的启动参数

参数	说明
<code>--insecure-bind-address=127.0.0.1</code>	HTTP（不安全）绑定的本机地址，0.0.0.0表示监听本机的所有地址，默认是127.0.0.1
<code>--insecure-port=8080</code>	不安全端口，HTTP（不安全）绑定的本机端口，默认是8080
<code>--bind-address=0.0.0.0</code>	HTTPS（安全）绑定的本机地址，默认是0.0.0.0
<code>--secure-port=6443</code>	安全端口，HTTPS（安全）绑定的本机端口，默认是6443，当为0时，不启动HTTPS
<code>--tls-cert-file=""</code>	HTTPS需要使用的x509证书，和--tls-private-key-file对应。当--tls-cert-file和--tls-private-key-file都为空的时候，会自动生成自签名证书和私钥，生成目录/var/run/kubernetes
<code>--tls-private-key-file=""</code>	HTTPS需要的x509私钥，和--tls-cert-file对应

Kubernetes API Server开启HTTPS需要准备的HTTPS证书，建议从权威CA机构申请：

```
--tls-cert-file=/path/to/cert
--tls-private-key-file=/path/to/key
--secure-port=6443
--bind-address=0.0.0.0
```

因为HTTP 是不安全的，所以应该限制HTTP 的访问，比如设置防火墙规格、防止不信任域的访问，其中最简单的方法是只允许本机访问：

```
--insecure-port=8080
--insecure-bind-address=127.0.0.1
```

10.2.2 认证与授权

Kubernetes API Server 目前支持认证（Authentication）和授权（Authorization）机制进行安全访问控制。认证和授权只作用于

Kubernetes API Server的安全端口（--secure-port），非安全端口（--insecure-port）不受约束。

10.2.2.1 认证

Basic Authentication

Basic Authentication是指客户端在使用HTTP访问受限资源时，必须使用用户名和密码以获取认证。这是认证中最简单的方法，长期以来，这种认证方法被广泛使用。当通过HTTP访问一个使用Basic Authentication保护的资源时，服务器通常会在HTTP请求的响应中加入一个“401需要身份验证”的头域，来通知客户提供用户凭证，以使用资源。如果正在使用浏览器访问需要认证的资源，浏览器会弹出一个窗口，让你输入用户名和密码，如果所输入的用户名在资源使用者的验证列表中，并且密码完全正确，此时，用户才可以访问受限的资源。但是这种方式安全性较低，就是简单地将用户名和密码进行Base64编码放到头域中。正是因为是Base64编码存储，最好使用HTTPS进行加密传输。

Kubernetes API Server开启Basic Authentication需要提供一个CSV格式的文件用于设置用户列表，每行表示一个用户，共3列：密码、用户名和用户ID，以下是一个示例：

basic_auth.csv

```
admin_passwd,admin,1  
test_passwd,test,2
```

然后设置Kubernetes API Server的启动参数:

```
--basic-auth-file=/path/to/basic_auth.csv
```

Token Authentication

Token是一个用户自定义的任意字符串，具有随机性、不可预测性，相比于密码更加安全，一般黑客或软件无法猜测出来。Token Authentication实际上同Basic Authentication类似，使用Token替换账号密码，可在一定程度上提高安全性。

Kubernetes API Server支持Token Authentication，同样需要提供一个CSV格式的文件用于设置用户列表，每行表示一个用户，共3列：Token、用户名和用户ID，以下是一个示例：

token_auth.csv

```
ceGlx8PkwDz0Z5Ls0e1DShpCD1j67r0a,admin,1  
5oEhn1YY6V7J31m3wZX1Tzp8XFX0AHXJ,test,2
```

其中Token是任意字符串，可以用以下命令生成：

```
$ echo $(cat /dev/urandom | base64 | tr -d "=+/" | dd bs=32  
count=1 2> /dev/null)
```

然后设置Kubernetes API Server的启动参数:

```
--token-auth-file =/path/to/token_auth.csv
```

Client Certificate Authentication

Client Certificate是一种用于证明用户身份的客户端数字证书，如果服务端开启Client Certificate Authentication，客户端访问的时候就需要提供Client Certificate。

Kubernetes API Server支持Client Certificate Authentication，需要提供信任的客户端CA证书，然后配置启动参数：

```
--client-ca-file=/path/to/ca.crt
```

OpenID Authentication

OpenID是一套以用户为中心的分散式身份认证系统，用户只需注册获取OpenID之后，就可以凭借此OpenID账号在多个系统之间自由登录使用，而不需要在每一个系统中注册账号，实现用户认证。

Kubernetes API Server支持OpenID Authentication，相关启动参数如下：

```
--oidc-issuer-url=<url>  
--oidc-client-id=<id_token>
```

Keystone Authentication

Keystone是OpenStack框架中负责管理身份验证、服务规则和服务令牌功能的模块。用户访问资源需要验证用户的身份与权限，服务执行操作也需要进行权限检测，这些都可以通过Keystone来处理。

Kubernetes API Server支持对接Keystone来实现认证，相关启动参数如下：

`--experimental-keystone-url=<AuthURL>`

10.2.2.2 授权

Kubernetes API Server在认证之后，通过授权（Authorization）可进一步进行安全访问控制。授权可以通过Kubernetes API Server的`--authorization-mode`启动参数设置以下几种模式：

- `--authorization-mode=AlwaysDeny`
- `--authorization-mode=AlwaysAllow`
- `--authorization-mode=ABAC`

其中`AlwaysDeny`表示拒绝所有请求，`AlwaysAllow`允许所有请求。

`ABAC`是一种基于属性的访问控制，根据设置好的访问策略检查请求的属性，不符合访问策略的请求会被拒绝。

`API`请求中有5个属性可以用作访问控制：

- 用户，请求用于认证的用户，比如使用`Basic Authentication`时，用户需要输入用户名和密码进行认证，那么授权则根据该用户进行判断。
- 用户组，用户所在的用户组，一个用户可以在多个用户组内。
- 请求是否只读，比如`GET`请求都是只读的。

- 请求访问的资源，比如/api/v1/namespaces/default/pods请求的资源是pods，一些情况下资源为空，比如/version。

- 资源所在的Namespace，某些资源不属于任何Namespace，比如Node，那么Namespace即为空。

当设置为ABAC模式时，还需要指定策略文件（--authorization-policy-file），策略文件采用一种One JSON Object Per Line的配置格式，即每一行是一个JSON格式的策略，用于设置访问控制权限，策略属性如表10-2所示。

表10-2 ABAC模式的策略属性

属性	类型	说明
user	string	如果指定，需要和请求认证用户匹配
group	string	如果指定，需要和请求认证用户组匹配
readonly	boolean	如果为true，说明只允许GET请求
resource	string	如果指定，需要和请求资源匹配
namespace	string	如果指定，需要和请求资源所在Namespace匹配

策略文件示例：

```
# admin拥有所有权限
```

```
{"user":"admin"}
```

```
# scheduler拥有读pods的权限
```

```
{"user":"scheduler","readonly": true,"resource": "pods"}
```

```
# scheduler拥有读写events的权限
```

```
{"user":"scheduler","resource": "events "}
```

```
# kubelet拥有读pods的权限
```

```
{"user":"kubelet","readonly": true,"resource": "pods"}

# kubelet拥有读services的权限
{"user":"kubelet","readonly": true,"resource": "services"}

# kubelet拥有读endpoints的权限
{"user":"kubelet","readonly": true,"resource": "endpoints"}

# kubelet拥有读写events的权限
{"user":"kubelet","resource": "events"}

# alice拥有namespace projectCaribou下的所有权限
{"user":"alice","namespace": "projectCaribou"}

# alice拥有namespace projectCaribou下的读权限
{"user":"bob","readonly": true,"namespace": "projectCaribou"}

# cindy拥有namespace projectCaribou下的pods的读权限
{"user":"cindy","resource": "pods","readonly":
true,"namespace": "projectCaribou"}
```

10.2.3 准入控制Admission Controller

请求在认证和授权之后，Kubernetes提供了Admission Controller进一步对请求进行准入控制。Admission Controller作为Kubernetes API

Server的一部分，并以插件的形式存在，不过需要编译进Kubernetes API Server可执行程序才能使用。

Kubernetes API Server将按顺序检查请求，如果其中任何一项没通过，那么就会拒绝请求。

Admission Controller支持的插件如表10-3所示。

表10-3 Admission Controller插件

名称	说明
AlwaysAdmit	此插件允许所有的请求
AlwaysDeny	此插件拒绝所有的请求
ServiceAccount	此插件用于支持Service Account
SecurityContextDeny	此插件检查创建Pod的Security Context 是否可用，不可用的话拒绝
ResourceQuota	此插件用于支持Resource Quota，拒绝对于任何超过Resource Quota 的请求
LimitRanger	用于支持Limit Ranger，拒绝不符合LimitRanger约束的请求
NamespaceLifecycle	删除一个Namespace会删除该Namespace下所有的资源（pods、services等）。在Namespace被删除的过程中，此插件将会拒绝任何试图在该Namespace下创建新资源的请求。除此之外，拒绝试图在不存在的Namespace下创建资源的请求

在 Kubernetes API Server 启动的时候，可以配置需要哪些 Admission Controller，以及它们的顺序，建议设置为：

--

```
admission_control=NamespaceLifecycle,LimitRanger,SecurityContextDeny,ServiceAccount,ResourceQuota
```

10.3 Service Account

Service Account概念的引入是基于这样的使用场景：运行在Pod里的进程需要调用Kubernetes API以及非Kubernetes API的其他服务。我

们使用Service Account来为Pod提供认证。

Service Account和User Account可能会带来一定程度上的混淆，它们的区别如下：

- User Account通常是为人设计的，而Service Account则是为运行在Pod中的应用设计的。
- User Account是全局的，即可以跨Namespace使用；而Service Account是限定Namespace的，即仅在所属的Namespace下使用。
- 创建一个新的User Account通常需要较高的特权并且需要经过比较复杂的业务逻辑，而Service Account则不然。

提示

开启Service Account功能，要添加Service Account Admission Controller，即设置Kubernetes API Server的启动参数：

`--admission_control=...ServiceAccount...`

Kubernetes API Server的相关启动参数如表10-4所示。

表10-4 Kubernetes API Server的启动参数

<code>--service-account-key-</code>	配置一个包含PEM-encoded x509 RSA的私钥或者公钥，用于验证Service Account Token
-------------------------------------	---

file=""	
---------	--

Kubernetes Controller Manager的相关启动参数如表10-5所示。

表10-5 Kubernetes Controller Manager的启动参数

--service-account-private-key-file=""	配置一个包含PEM-encoded x509 RSA的私钥，用于签发Service Account Token
--root-ca-file=""	配置访问Kubernetes API Server的CA 证书，将赋值给Service Account Secret

10.3.1 使用默认Service Account

Kubernetes Controller Manager会为每个Namespace默认创建一个Service Account，我们称为默认Service Account：

```
$ kubectl get serviceaccount
```

```
NAME          SECRETS   AGE
default       1         1d
```

默认Service Account包含一个Secret：

```
$ kubectl describe serviceaccount default
```

```
Name:                default
Namespace:           default
Labels:              <none>
```

Mountable secrets: default-token-7t9ja

Tokens: default-token-7t9ja

Image pull secrets: <none>

查询该Secret:

```
$ kubectl describe secret default-token-7t9ja
```

Name: default-token-7t9ja

Namespace: default

Labels: <none>

Annotations: kubernetes.io/service-account.name=default, kubernetes.io/service-account.uid=...

Type: kubernetes.io/service-account-token

Data

====

token: ...

ca.crt: 1838 bytes

查询显示这是一个 kubernetes.io/service-account-token 类型的 Secret，称为Service Account Secret，包含两个数据token和ca.crt，这也是由Kubernetes Controller Manager生成的。其中token是由--service-account-private-key-file指定的密钥签发生成的，而ca.crt是由--root-ca-file指定的CA证书。

这样一来，新建的Pod如果没有指定Service Account，就会使用默认 Service Account，挂载 Service Account Secret 到容器目录中（/var/run/secrets/kubernetes.io/serviceaccount），Pod中的应用通过token和ca.crt访问Kubernetes API Server。同时Kubernetes提供非常方便的方式让Pod获取Kubernetes API Server的地址。Kubernetes在初始化的时候会创建一个Kubernetes Service，叫作kubernetes，它代表的就是Kubernetes API Server：

Kubernetes在初始化的时候会创建一个Kubernetes Service，叫作kubernetes，它代表的就是Kubernetes API Server：

```
$ kubectl describe service kubernetes
```

```
Name:      kubernetes
```

```
Namespace: default
```

```
Labels:     component=apiserver, provider=kubernetes
```

```
Selector:   <none>
```

```
Type:      ClusterIP
```

```
IP:        10.254.0.1
```

```
Port:      https 443/TCP
```

```
Endpoints: 192.168.3.146:6443
```

```
Session Affinity: None
```

```
No events.
```

Kubernetes Service的虚拟IP是10.254.0.1，这是Kubernetes API Server 配置的 Service 网段的第一个IP（--service-cluster-ip-range=10.254.0.0/16），然后将会转发HTTPS的443端口到192.168.3.146:6443，即Kubernetes API Server的地址和HTTPS安全端

口。并且Kubernetes Service是最早创建的，新建的Pod中可以通过环境变量获取来访问Kubernetes API Server（也可以通过DNS方式）：

```
KUBERNETES_SERVICE_HOST=10.254.0.1
KUBERNETES_PORT_443_TCP=tcp://10.254.0.1:443
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_ADDR=10.254.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_PORT=tcp://10.254.0.1:443
KUBERNETES_PORT_443_TCP_PROTO=tcp
```

Pod中的进程就可以访问Kubernetes API Server，我们现在简单实现一个脚本curl-k8s-api.sh来调用 Kubernetes API：

```
#!/bin/sh
```

```
# curl-k8s-api.sh
```

```
Endpoint=$1
```

```
ServiceAccountToken=$(cat
```

```
/var/run/secrets/kubernetes.io/serviceaccount/token)
```

```
ServiceAccountRootCA=/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
```

```
KubernetesServerURL="https://$KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_PORT"
```

```
curl -XGET -H "Authorization: Bearer $ServiceAccountToken" \
```



```
--cacert $ServiceAccountRootCA \  
${KubernetesServerURL}${Endpoint}
```

脚本curl-k8s-api.sh读取Service Account的token和ca.crt，通过环境变量生成URL，然后通过curl命令调用Kubernetes API Server。我们需要将脚本curl-k8s-api.sh放入Pod的容器中，然后访问/api获取版本信息：

```
$ kubectl exec pod -- curl-k8s-api.sh /api  
{  
  "versions": [  
    "v1"  
  ]  
}
```

另外，Service Account会自动创建一个认证用户，用户的命名格式是：

```
system:serviceaccount:[namespace]:[serviceaccountname]
```

比如对于在Namespace my-ns中，默认Service Account对应的用户名是：

```
system:serviceaccount:my-ns:default
```

如果Kubernetes API Server设置了ABAC的授权模式，需要为Service Account设置授权策略，否则会导致Pod无法通过Service Account访问Kubernetes API Server，比如设置为只读权限：

```
{"user": "system:serviceaccount:    my-ns:default",    "readonly": true}
```

10.3.2 创建自定义Service Account

用户可以创建自定义的Service Account，Service Account的定义文件serviceaccount.yaml:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-serviceaccount
```

通过定义文件创建Service Account:

```
$ kubectl create -f serviceaccount.yaml
serviceaccount "my-serviceaccount" created
```

查询Service Account的详细信息:

```
$ kubectl describe serviceaccount my-serviceaccount
Name:    my-serviceaccount
Namespace:            default
Labels:    <none>

Image pull secrets: <none>
```

Mountable secrets: my-serviceaccount-token-ivsm1

Tokens: my-serviceaccount-token-ivsm1

可以看到，该Service Account包含一个Service Account Secret，Kubernetes Controller Manager会保证每个Service Account都至少包含一个Service Account Secret。如果不希望使用自动创建的Service Account Secret，可以手工创建，Service Account Secret的定义文件serviceaccountSecret.yaml:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-serviceaccount-secret
  annotations:
    kubernetes.io/service-account.name: my-serviceaccount
type: kubernetes.io/service-account-token
```

通过定义文件创建Service Account Secret:

```
$ kubectl create -f serviceaccountSecret.yaml
secret "my-serviceaccount-secret" created
```

添加创建好的Service Account Secret到Service Account:

```
$ kubectl patch serviceaccount my-serviceaccount \
-p '{"secrets": [{"name": "my-serviceaccount-secret"}]}'
"my-serviceaccount" patched
```

然后删除自动创建的Service Account Secret:

```
$ kubectl delete secret my-serviceaccount-token-ivsm1  
secret "my-serviceaccount-token-ivsm1" deleted
```

这样就成功替换了Service Account Secret:

```
$ kubectl describe serviceaccount my-serviceaccount
```

```
Name: my-serviceaccount
```

```
Namespace: default
```

```
Labels: <none>
```

```
Image pull secrets: <none>
```

```
Mountable secrets: my-serviceaccount-secret
```

```
Tokens: my-serviceaccount-secret
```

在Pod的定义中可以通过.spec.serviceAccountName指定Service Account, 如下所示:

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  name: busybox
```

```
  namespace: default
```

```
spec:
```

```
  containers:
```

```
- name: busybox
  image: busybox
  command:
    - sleep
    - "3600"
  restartPolicy: Always
  serviceAccountName: my-serviceaccount
```

10.3.3 Service Account添加Image Pull Secret

在Service Account中添加Image Pull Secret时，当Pod使用该Service Account的时候就自动关联上其中的Image Pull Secret；而当为默认Service Account添加Image Pull Secret时，那么Pod会自动关联上Image Pull Secret，不必显示配置。

首先创建一个Image Pull Secret，可参考4.3.1节：

```
$ kubectl get secret myregistrykey
```

NAME	TYPE		DATA	AGE
myregistrykey	kubernetes.io/dockercfg	1	1h	

添加Image Pull Secret到默认Service Account：

```
$ kubectl patch serviceaccounts default \
-p '{"imagePullSecrets": [{"name": "myregistrykey"}]}'
"default" patched
```

修改默认Service Account成功后，可以查询到Image Pull Secret添加成功：

```
$ kubectl describe serviceaccounts default
```

```
Name: default
```

```
Namespace: default
```

```
Labels: <none>
```

```
Image pull secrets: myregistrykey
```

```
Mountable secrets: default-token-mymp9
```

```
Tokens: default-token-mymp9
```

这样一来，新创建的Pod都会自动关联上Image Pull Secret:

```
spec:
```

```
  imagePullSecrets:
```

```
  - name: myregistrykey
```

10.4 容器安全

10.4.1 Linux Capability

Docker对于容器中的root用户是进行限制的，这是基于Linux内核的Capability机制实现的。Linux Capability将内核系统资源访问划分为许多的权限属性，从而可以对权限进行精细化管理。

对于Docker容器中的root用户，很多系统相关的操作权限都是被剥夺的，只具备超级用户的一些基本权限。如果需要授予Docker容器足够的权限，可以使用特权模式，即设置docker run的参数--privileged=true。

在 Pod 的定义中可以通过 .spec.containers[].securityContext.privileged=true 设置容器的特权模式：

```
apiVersion: v1
kind: Pod
metadata:
  name: nfs-server
  labels:
    role: nfs-server
spec:
  containers:
    - name: nfs-server
      image: jsafrane/nfs-data
      ports:
        - name: nfs
          containerPort: 2049
      securityContext:
        privileged: true
```

注意

Pod中的容器要设置特权模式，需要设置Kubernetes API Server和Kubelet的启动参数--allow-privileged=true来允许开启容器的特权模式。

当Docker容器设置了特权模式之后，那么Docker容器的root权限将得到大幅度提升。由于Docker容器与宿主机处于共享同一个内核操作系统的状态，因此Docker容器将完全拥有内核的管理权限，这就存在安全隐患。

而且对应用程序来说，往往只需要特定的权限，比如一个程序需要使用ping命令，这是一个SUID命令，会以root权限运行，而实际上这个程序只是需要RAW套接字建立必要ICMP数据包，除此之外的其他root权限对这个程序都是没有必要的。

Docker支持添加和删除容器的Linux Capability，即docker run命令的 --cap-add 和 --cap-drop 参数。在 Pod 的定义中可以通过 .spec.containers[].securityContext.capabilities.add 和 .spec.containers[].securityContext.capabilities.drop 添加和删除容器的Linux Capability:

```
apiVersion: v1
kind: Pod
metadata:
  name: nfs-server2
  labels:
    role: nfs-server
spec:
```



```
containers:
  - name: nfs-server
    image: nginx
    ports:
      - name: nfs
        containerPort: 2049
    securityContext:
      capabilities:
        add:
          - IPC_LOCK
          - NET_ADMIN
        drop:
          - SETGID
```

10.4.2 SELinux

SELinux (Security Enhanced Linux) 是Linux内核的安全模块，提供了一种灵活的强制访问控制系统。SELinux拥有一个灵活而强制性的访问控制结构，旨在提高Linux系统的安全性，提供强健的安全保证。

Docker支持使用SELinux进行安全加固，通过SELinux可以根据类别和MCS/MLS等级来区分进程，保护宿主机不受容器影响。

docker run中通过--security-opt可以修改容器的MCS/MLS标签：

```
--security-opt="label:user:USER"  
--security-opt="label:role:ROLE"  
--security-opt="label:type:TYPE"  
--security-opt="label:level:LEVEL"
```

相应的，在Pod的定义中可按如下方式修改容器的MCS/MLS标签：

```
securityContext:  
  seLinuxOptions:  
    user: USER  
    role: ROLE  
    type: TYPE  
    level: LEVEL
```

10.5 多租户

多租户是云计算中的一个重要能力，是云计算集中式的数据中心，以服务的形式提供给用户。多租户是共享和隔离互相作用下的产物，用户需要处在不同的租户下，同租户内部是共享的，但是不同租户之间是隔离的。

Kubernetes中的Namespace就是租户的概念，Kubernetes整个平台的内容通过Namespace划分为多个逻辑平面，不同个人或者团队在不同Namespace中共享Kubernetes，Namespace之间互相不感知。

Kubernetes以Namespace作为管理单位，可以控制安全访问策略，设置资源配额等，为此需要使用Kubernetes规划好Namespace，比如可

可以根据功能、区域、部门或者业务等。

我们可以像其他API对象一样创建Namespace，Namespace定义文件development- namespace.yaml:

```
apiVersion: v1
kind: Namespace
metadata:
  name: development
  labels:
    name: development
```

通过定义文件创建Namespace:

```
$ kubectl create -f development-namespace.yaml
namespace "development" created
```

创建成功后可以查询Namespace:

```
$ kubectl describe namespace development
Name: development
Labels:  name=development
Status:  Active
```

No resource quota.

No resource limits.

第11章

Kubernetes资源管理

资源管理是Kubernetes的一个关键能力，Kubernetes需要为应用分配足够的资源，又要防止应用无限制使用资源，随着应用规模数量级的增加，这些问题就显得至关重要。本章将阐述Kubernetes资源管理的模型和具体实现方法，包括资源分配策略和配额限制等。

11.1 Kubernetes资源模型

虚拟化技术是云计算平台的基础，其目标是对计算资源进行整合或划分，这是云计算管理平台中的关键技术。虚拟化技术为云计算管理平台的资源管理提供了资源调配上的灵活性，从而使得云计算管理平台可以通过虚拟化层整合或划分计算资源。

相比于虚拟机，新出现的容器技术使用了一系列的系统级别的机制，诸如利用Linux Namespace进行空间隔离，通过文件系统的挂载点决定容器可以访问哪些文件，通过Cgroup确定每个容器可以利用多少资源。此外，容器之间共享同一个系统内核，这样当同一个内核被多个容器使用时，内存的使用效率会得到提升。

容器和虚拟机两大虚拟化技术，虽然实现方式完全不同，但是它们的资源需求和模型其实是类似的。容器像虚拟机一样需要内存、CPU、硬盘空间和网络带宽，宿主机系统可以将虚拟机和容器都视作一个整体，为这个整体分配其所需的资源，并进行管理。当然，虚拟机提供了

专用操作系统的安全性和更牢固的逻辑边界，而容器在资源边界上比较松散，这带来了灵活性以及不确定性。

Kubernetes是一个容器集群管理平台，Kubernetes需要统计整体平台的资源使用情况，合理地将资源分配给容器使用，并且要保证容器生命周期内有足够的资源来保证其运行。更进一步，如果资源发放是独占的，即资源已发放给了一个容器，同样的资源不会发放给另外一个容器，对于空闲的容器来说占用着没有使用的资源（比如CPU）是非常浪费的，Kubernetes需要考虑如何在优先度和公平性的前提下提高资源的利用率。

11.2 资源请求和限制

计算资源是Pod或者容器运行所需的，包括：

- CPU

单位是核（core）。

cpu: 1 #1核

cpu: 0.25 #0.25核

cpu: 250m #0.25核

- 内存（Memory）

单位是字节（byte）。

memory: 1024 #1024 Byte

memory: 512Ki #512 KByte

```
memory: 256Mi #256 MByte
```

```
memory: 1.5Gi #1.5 GByte
```

创建Pod的时候，可以指定每个容器的资源请求（Request）和资源限制（Limit），资源请求是容器所需的最小资源需求，资源限制则是容器不能超过的资源上限，它们的大小关系必须是：

$$0 \leq \text{request} \leq \text{limit} \leq \text{Infinity}$$

在容器的定义中，资源请求通过resources.requests设置，资源限制通过resources.limits设置，目前可以指定的资源类型只有CPU和内存。资源请求和限制是可选配置，默认值根据是否设置Limit Range而定。如果资源请求没有指定也没有默认值，那么资源请求就等于资源限制。

以下定义的Pod包含两个容器：第一个容器的资源请求是0.5核CPU和256 MByte内存，资源限制是1核CPU和512 MByte内存；第二个容器的资源请求是0.25核CPU和128MByte内存，资源限制是1核CPU和512 MByte内存：

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
```

```
        memory: "256Mi"
        cpu: "500m"
    limits:
        memory: "512Mi"
        cpu: "1000m"
- name: wp
  image: wordpress
  resources:
    requests:
        memory: "128Mi"
        cpu: "250m"
    limits:
        memory: "512Mi"
        cpu: "1000m"
```

Pod的资源请求/限制是Pod中所有容器资源请求/限制的和，比如该Pod的资源请求是0.75核CPU和384MByte内存，资源限制是2核CPU和1024MByte内存。

Kubernetes在调度Pod的时候，Pod的资源请求是调度的一个关键指标。Kubernetes会获取Kubernetes Node的最大资源容量（通过cAdvisor接口），并计算出已使用的资源情况，比如Node能够容纳2核CPI和2GByte内存，Node上已经运作了4个Pod，共请求1.5核CPU和1GByte内存，剩余0.5核CPU和1GByte内存。Kubernetes Scheduler调度Pod的时候会检查Node上是否还有足够资源来满足Pod的资源请求，不满足则将该Node排除。

资源请求能够保证Pod有足够的资源来运行，而资源限制则是防止某个Pod无限制地使用资源，导致其他Pod崩溃。特别是在公有云场景，往往会有恶意软件通过抢占内存来攻击平台。

Docker容器是使用Linux Cgroup来实现资源限制的，`docker run`就提供了参数对CPU和内存进行限制。

- `--memory`

`docker run`通过`--memory`给一个容器可用的内存配额，Cgroup会限制容器的内存使用，一旦超过配额，容器将会被终止。

Kubernetes中Docker容器的`--memory`值就是`resources.limits.memory`的值，比如`resources.limits.memory=512Mi`，那么`--memory`的值就是 $512 \times 1024 \times 1024 \times 1024$ 。

- `--cpu-shares`

`docker run`通过`--cpu-shares`设置一个容器的可用的CPU配额。需要特别注意的是，这是一个相对权重，与实际的处理速度无关。每个新的容器默认将有1024 CPU配额，当我们单独讲它的时候，这个值并不意味着什么。但是如果启动两个容器并且两个都将使用100%的CPU，CPU时间将在这两个容器之间平均分割，因为它们两个都有同样的CPU配额。如果我们设置容器的CPU配额是512，相对于另外一个1024 CPU配额容器，它将使用1/3的CPU时间。但这并不意味着它仅仅能使用1/3的CPU时间。如果另外一个容器（1024 CPU配额的）是空闲的，其他容器将被允许使用100% CPU。对于CPU来说，难以清楚地说明多少CPU被分配给了哪个容器，这取决于实际的运行情况。

Kubernetes中Docker容器的--cpu-shares值通过resources.requests.cpu或者resources.requests.cpu乘以1024而得。如果指定resources.requests.cpu,那么--cpu-shares就等于resources.requests.cpu乘以1024,如果没有指定resources.requests.cpu,但是指定了resources.limits.cpu,那么--cpu-shares就等于resources.limits.cpu乘以1024,如果resources.limits.cpu和resources.limits.cpu都没有指定,那么resources.requests.cpu就取最小值(当前版本最小值为2),具体如下:

```
if resources.requests.cpu is defined
    cpuShares=resources.requests.cpu * 1024
else if resources.requests.cpu is undefined
    if resources.limits.cpu is defined
        cpuShares=resources.limits.cpu * 1024
else if resources.limits.cpu is defined
    cpuShares = minShares
```

比如resources.requests.cpu=250m,那么--cpu-share的值就是250m*1024=256。

11.3 Limit Range

Limit Range设计的初衷是为了满足以下场景:

- 能够约束租户的资源需求。
- 能够约束容器的资源请求范围。
- 能够约束Pod的资源请求范围。

- 能够指定容器的默认资源限制。
- 能够指定Pod的默认资源限制。
- 能够约束资源请求和限制之间的比例。

提示

Kubernetes 要开启 Limit Range 功能，首先要添加 Limit Range Admission Controller，即设置Kubernetes API Server的启动参数：

```
--admission_control=...LimitRange...
```

Limit Range包含两种类型的设置： Container和Pod，包含约束和默认值的配置，如表11-1和表11-2所示。

表11-1 Limit Range Container配置

类型	Container
资源类型	memory cpu
约束	min: min <= Request (required) <= Limit (optional) max: Limit (required) <= max

	maxLimitRequestRatio: $\text{maxLimitRequestRatio} \leq (\text{Limit}(\text{required}, \text{non-zero}) / \text{Request}(\text{required}, \text{non-zero}))$
默认值	default: Limit的默认值 defaultRequest: Request的默认值

表11-2 Limit Range Pod配置

类型	Pod
资源类型	memory cpu
约束	min: $\text{min} \leq \text{Request}(\text{required}) \leq \text{Limit}(\text{optional})$ max: $\text{Limit}(\text{required}) \leq \text{max}$ maxLimitRequestRatio: $(\text{Limit}(\text{required}, \text{non-zero}) / \text{Request}(\text{required}, \text{non-zero})) \text{ maxLimitRequestRatio}$
默认值	Pod的默认值无须直接配置，根据容器的默认值而得

创建Limit Range的时候需要注意以下几点。

- 配置数值的时候需要满足以下条件

`Min (if specified) <= DefaultRequest (if specified) <= Default (if specified) <= Max (if specified)`

- 默认值行为

当Default未设置:

```
if LimitRangeItem.Default[resourceName] is undefined
  if LimitRangeItem.Max[resourceName] is defined
    LimitRangeItem.Default[resourceName] =
LimitRangeItem.Max[resourceName]
```

当DefaultRequest未设置:

```
if LimitRangeItem.DefaultRequest[resourceName] is undefined
  if LimitRangeItem.Default[resourceName] is defined
    LimitRangeItem.DefaultRequest[resourceName] =
LimitRangeItem.Default[resourceName]
  else if LimitRangeItem.Min[resourceName] is defined
    LimitRangeItem.DefaultRequest[resourceName] =
LimitRangeItem.Min[resourceName]
```

当在Namespace下创建Limit Range后，就可以设置Pod或者容器的资源请求和限制默认值，更重要的功能是对Pod和容器的资源规格配置

进行约束。现在我们在Namespace development下创建一个Limit Range, Limit Range的定义文件limits.yaml:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limits
  namespace: development
spec:
  limits:
    - type: Pod
      max:
        cpu: 4
        memory: 2Gi
      min:
        cpu: 250m
        memory: 8Mi
      maxLimitRequestRatio:
        cpu: 4
        memory: 4
    - type: Container
      default:
        cpu: 500m
        memory: 512Mi
      defaultRequest:
        cpu: 250m
        memory: 256Mi
```

```

max:
  cpu: 2
  memory: 1Gi
min:
  cpu: 250m
  memory: 8Mi
maxLimitRequestRatio:
  cpu: 2
  memory: 2

```

通过定义文件创建Limit Range:

```
$ kubectl create -f limits.yaml
```

```
limitrange "limits" created
```

创建成功, 查询Limit Range:

```
$ kubectl describe limitranges limits --namespace=development
```

```
Name: limits
```

```
Namespace: development
```

Type	Resource	Min	Max	Request	Limit	Limit/Request
Pod	cpu	250m	4	-	-	4
Pod	memory	8Mi	2Gi	-	-	4
Container	cpu	250m	2	250m	500m	2
Container	memory	8Mi	1Gi	256Mi	512Mi	2

对于以上内容有以下几点说明。

- 默认值

1. 一个容器的默认资源请求是256Mbyte内存和250m核CPU。
2. 一个容器的默认资源限制是512Mbyte内存和500m核CPU。

- 约束

1. 一个Pod的资源请求和限制必须满足:

```
250m <= requests.cpu <= limits.cpu <= 4
8Mi <= requests.memory <= limits.memory <= 2Gi
limits.memory/requests.memory <= 4
limits.cpu/requests.cpu <= 4
```

2. 一个容器的资源请求和限制必须满足:

```
250m <= requests.cpu <= limits.cpu <= 2
8Mi <= requests.memory <= limits.memory <= 1Gi
limits.memory/requests.memory <= 2
limits.cpu/requests.cpu <= 2
```

至此，在 Namespace development 中创建的 Pod 都需要满足以上约束，比如创建一个Pod，Pod的定义文件test-pod.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: test
  namespace: development
```

spec:

containers:

- name: container1

image: nginx

resources:

requests:

memory: 512Mi

cpu: "999m"

limits:

memory: "512Mi"

cpu: "2"

- name: container2

image: nginx

resources:

requests:

memory: 512Mi

cpu: "250m"

limits:

memory: "1Gi"

cpu: "500m"

- name: container3

image: nginx

resources:

requests:

memory: 8ki

cpu: "250m"

limits:


```
memory: "1Gi"
cpu: "500m"
```

创建Pod会失败:

```
$ kubectl create -f test-pod.yaml
```

```
Error from server: error when creating "test-pod.yaml": Pod
"test" is forbidden: [
Maximum memory usage per Pod is 2Gi, but limit is 2684354560.,
cpu max limit to request ratio per Container is 2, but provided
ratio is 2.002002.,
Minimum memory usage per Container is 8Mi, but request is 8Ki.,
memory max limit to request ratio per Container is 2, but
provided ratio is 131072.000000.]
```

其中有4个错误:

1. Pod 的最大内存为 2Gi，但是创建的 Pod 资源限制是 1Gi+1Gi+512Mi=2684354560。
2. 容器的CPU限制和请求比例不能超过2，容器container1的CPU限制和请求比例为2/999m=2.002002。
3. 容器的最小内存是8mi，容器container3的CPU请求是8Ki。
4. 容器的内存限制和请求比例不能超过2，容器container3的内存限制和请求比例为500m /8ki =131072.000000。

11.4 Resource Quota

Kubernetes 是一个多租户架构，当多用户或者团队共享一个 Kubernetes 系统的时候，系统管理员需要防止租户的资源强占，定义好资源分配策略。比如 Kubernetes 系统共有 20 核 CPU 和 32 GByte 内存，分配给 A 租户 5 核 CPU 和 16 GByte，分配给 B 租户 5 核 CPU 和 8 GByte，预留 10 核 CPU 和 8 GByte 内存。这样，租户中所使用的 CPU 和内存的总和不能超过指定的资源配额，促使其更合理地使用资源。

Kubernetes 中提供 API 对象 Resource Quota（资源配额）来实现资源配额，Resource Quota 不仅可以作用于 CPU 和内存，另外还可以限制比如创建 Pod 的数目。Resource Quota 支持的类型如表 11-3 和表 11-4 所示。

• 计算资源配额

表11-3 计算资源配额

资源名称	说明
cpu	CPU配额
memory	内存配额

限制计算资源的使用，Namespace 中所有 Pod 的资源请求总和不能超过配额，比如 Namespace A 的内存配额为 1 GByte，那么 Namespace A 中所有 Pod 的 Memory 请求总和，即 resources.requests.memory 的总和不能超过 1 GByte。

• Kubernetes API对象资源配额

表11-4 Kubernetes API对象资源配额

资源名称	说明
pods	Pod的总数目
services	Service的总数目
replicationcontrollers	Replication Controller的总数目
resourcequotas	Resource Quota的总数目
secrets	Secret的总数目
persistentvolumeclaims	Persistent Volume Claim的总数目

限制Kubernetes API对象的创建数目，Namespace中API对象的创建数目不能超过配额，比如限制Namespace A的Pod和Service的创建数目。

提示

Kubernetes 要开启 Resource Quota 支持，首先要添加 Resource Quota Admission Controller，即设置Kubernetes API Server的启动参数：

```
--admission_control=...ResourceQuota...
```

默认情况下，Namespace是没有Resource Quota的，需要另外创建Resource Quota。一般情况下，一个Namespace下配置一个Resource Quota即可，但是可以创建多个Resource Quota，将按多个Resource Quota最小值作为配额值。比如一个Resource Quota设置Pod的数目配额为10，另一个Resource Quota设置Pod的数目配额为5，那么Pod的数目不能超过5。

现在我们在Namespace development下创建一个Resource Quota，Resource Quota的定义文件quota.yaml：

```

apiVersion: v1
kind: ResourceQuota
metadata:
  name: quota
  namespace: development
spec:
  hard:
    cpu: "20"
    memory: 50Gi
    persistentvolumeclaims: "10"
    pods: "10"
    replicationcontrollers: "20"
    resourcequotas: "1"
    secrets: "10"
    services: "5"

```

通过定义文件创建Resource Quota:

```

$ kubectl create -f quota.yaml
resourcequota "quota" created

```

创建成功后可以查询Resource Quota，其中包括配额值和使用值:

```

$ kubectl describe resourcequota quota --namespace=development

```

```

Name:      quota
Namespace:  development
Resource           Used  Hard
-----
cpu               0    20

```

```
memory      0  1Gi
persistentvolumeclaims 0  10
pods        0  10
replicationcontrollers 0  20
resourcequotas 1  1
secrets      1  10
services                    0  5
```

一旦Namespace有Resource Quota，创建Pod的时候就必须指定资源请求，否则Pod会创建失败（Kubernetes返回403 FORBIDDEN）。同样的，Pod请求的资源超过了资源配额也会创建失败（Kubernetes将返回403 FORBIDDEN）。

现在在Namespace development下创建一个Pod，请求0.25核CPU和128MByte的内存：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: development
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
      protocol: TCP
```

```
resources:
  requests:
    memory: "128Mi"
    cpu: "250m"
  limits:
    memory: "512Mi"
    cpu: "500m"
```

在Pod创建成功后，可以查询到相应的资源使用情况，共消耗1个Pod，以及0.25核CPU和128MByte的内存：

```
$ kubectl describe resourcequota quota --namespace=development
```

```
Name:      quota
Namespace:  development
Resource           Used    Hard
-----
cpu               250m    20
memory           134217728    1Gi
persistentvolumeclaims 0    10
pods              1    10
replicationcontrollers 0    20
resourcequotas    1    1
secrets           1    10
services          0    5
```

第12章

管理和运维Kubernetes

Kubernetes向下接管底层资源，向上托管业务应用，要管理和运维这样一套系统可以说是一个巨大的挑战。幸运的是，Kubernetes已经提供了一些很好的支持来帮助管理Kubernetes。本章将介绍Kubernetes的高可靠性方案、平台监控、平台日志等。

12.1 Daemon Pod

从系统管理的需求来说，我们常常需要在Kubernetes的所有节点上运行一些守护进程（Daemon），比如日志收集、监控等。传统的做法是使用一些Init工具（比如init、upstartd，或者systemd）来进行管理，而Kubernetes中支持以Pod的形式运行这些守护进程，我们称为Daemon Pod，实现的方式包括Static Pod和Daemon Set。

12.1.1 Static Pod

Static Pod是直接由Kubelet组件创建、运行在Node上的Pod，Kubelet组件负责Static Pod的持续运行，而无须Replication Controller进行关联管理。这样一来，Static Pod就同Kubelet进行绑定，从而运行在Node上作为Daemon Pod。

Static Pod的创建是通过在Kubelet指定的Manifest目录中放入Pod的定义文件（JSON或者YAML格式）进行的，Manifest目录是通过Kubelet的启动参数--config配置的。本书Kubernetes运行环境指定Kubelet的Manifest目录为/etc/kubernetes/manifests。

现在通过Static Pod在Kubernetes Node上运行Prometheus，一个开源的服务监控系统，可以实现对Docker容器进行监控。这需要在所有Node上的Manifest目录中创建Static Pod的定义文件prometheus-node-exporter.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: prometheus-node-exporter
  labels:
    daemon: prom-node-exp
spec:
  containers:
  - name: c
    image: prom/prometheus
    ports:
      - name: serverport
        containerPort: 9090
        hostPort: 9090
```

Kubelet将在Node上运行Static Pod，其中Static Pod的名称是Pod名称拼接上Node的名称：[pod_name]-[node_name]:


```
$ kubectl get pod --selector daemon=prom-node-exp --output wide
```

NAME			READY	STATUS
RESTARTS	AGE	NODE		
		prometheus-node-exporter-kube-node-1	1/1	Running 0
15s		kube-node-1		
		prometheus-node-exporter-kube-node-2	1/1	Running 0
18s		kube-node-2		
		prometheus-node-exporter-kube-node-3	1/1	Running 0
25s		kube-node-3		

如果删除该Static Pod，Kubelet会重新创建Static Pod，保证其持续运行，而只有在Manifest目录中删除该Static Pod的定义文件，Static Pod才会被删除。

除了直接在Manifest目录中放入Static Pod定义文件，还可以通过Kubelet的启动参数--manifest-url=<URL>指定远程URL，Kubelet将定期下载Static Pod的定义文件进行创建或者更新。

使用Static Pod能够运行Daemon Pod，但是Static Pod的管理是比较低效的。比如发生变更，就可能需要修改每个Node上的Kubelet配置。为此，Kubernetes提供了一个更加强大的机制来管理运行Daemon Pod。

12.1.2 Daemon Set

Kubernetes提供了Daemon Set，用来在Kubernetes Node上创建运行Daemon Pod。Daemon Set的作用同Replication Controller类似，它们都

是管理控制Pod的，只不过Daemon Set是保证所有（或者部分）Node上都能够运行Daemon Pod。

提示

在当前版本（Kubernetes v1.1.1）中，Daemon Set 是 Beta 测试阶段，需要设置 Kubernetes API Server 启动参数 `--runtime-config=extensions/v1beta1/daemonsets=true` 来开启Daemon Set支持。

现在通过 Daemon Set 在 Kubernetes Node 上运行 Prometheus，Daemon Set的定义文件prometheus-node-exporter.yaml:

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: prometheus-node-exporter
spec:
  template:
    metadata:
      labels:
        daemon: prom-node-exp
    spec:
      containers:
        - name: c
          image: prom/prometheus
```

```
ports:
- name: serverport
  containerPort: 9090
  hostPort: 9090
```

可以看出，Daemon Set的定义和Replication Controller类似，通过.spec.template设置Pod的模板。因为Daemon Pod需要持续运行，所以Pod的模板中的重启策略只能是Always。

在默认情况下，Daemon Set会在所有Node上运行Daemon Pod，通过.spec.template.spec. nodeSelector设置Node Selector，可以只在匹配的Node上运行Daemon Pod。

通过定义文件创建Daemon Set:

```
$ kubectl create -f prometheus-node-exporter.yaml --
validate=false
```

```
daemonset "prometheus-node-exporter" created
```

```
$ kubectl get daemonset prometheus-node-exporter
```

NAME	CONTAINER(S)	IMAGE(S)
SELECTOR	NODE-SELECTOR	
prometheus-node-exporter	c	prom/prometheus
daemon=prom-node-exp	<none>	

Demon Set创建成功后，将在所有Node上运行Pod:

```
$ kubectl get pods --selector daemon=prom-node-exp --output
wide
```

NAME			READY	STATUS
RESTARTS	AGE	NODE		
prometheus-node-exporter-qpta1	1/1	Running	0	
50s	kube-node-1			
prometheus-node-exporter-j04ks	1/1	Running	0	
50s	kube-node-2			
prometheus-node-exporter-ss23x	1/1	Running	0	
50s	kube-node-3			

当删除Daemon Set后，相关联的Pod也会被删除：

```
$ kubectl delete daemonset prometheus-node-exporter
daemonset "prometheus-node-exporter" deleted
```

```
$ kubectl get pods --selector daemon=prom-node-exp -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	NODE
------	-------	--------	----------	-----	------

如果希望删除Daemon Set而保留关联的Pod，运行kubectl delete时加上参数--cascade=false即可。

12.2 Kubernetes的高可用性

高可用性是一个老生常谈的话题，当然是因为这是人们非常关心的特性，它决定了一个系统的最终价值。在生产环境中，任何系统都需要持续可靠地运行，保持其服务的高度可用性，这是一个最基本的要求。特别是对于Kubernetes这样的云平台，一个将要承载着大量应

用的系统，它的任何故障都可能大面积地影响业务，其重要性自然不言而喻。

Kubernetes属于典型的主从分布式架构，Kubernetes的重要数据集中存储在Etcd，数据层的可靠性至关重要，对Etcd进行集群化是必不可少的（可参考14.3.2节）。

Kubernetes Node作为Pod的运行机，原生支持集群化扩展来提供容灾容错能力。Kubernetes Node运行后将会注册到Kubernetes Master，并定时上报心跳信息以说明其可用。Kubernetes Master调度Pod到可用的Kubernetes Node部署运行，如果有Kubernetes Node发生宕机，Kubernetes Master会将该Kubernetes Node设置为不可用状态。然后Replication Controller会重新创建Pod，从而调度到新的Kubernetes Node上。当然，Kubernetes Node的数目至少要大于2才可以保障Pod的高可用性。

在Kubernetes Master的高可用性方案中需要特别注意，Kubernetes API Server可以多实例运行，而Kubernetes Scheduler和Kubernetes Controller Manager不能有多个实例同时运行。Kubernetes官方提供了一种方案，如图12-1所示。

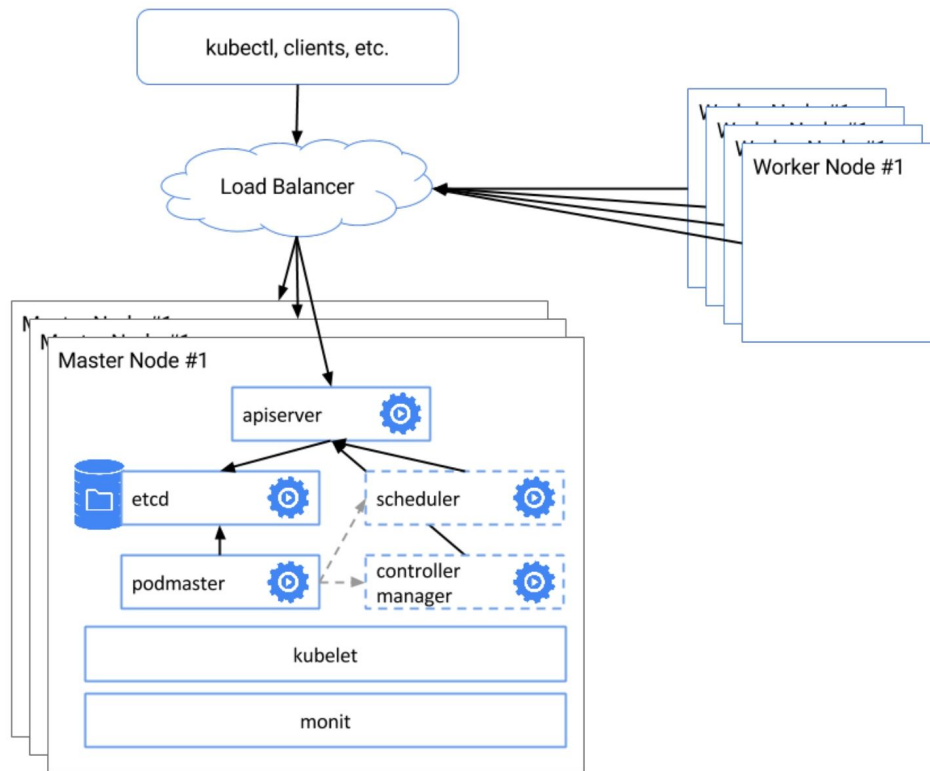


图12-1 Kubernetes的高可用性

在这个方案中，Kubernetes Master多节点部署，其中Kubernetes API Server、Kubernetes Scheduler和Kubernetes Controller Manager 3个组件分别作为Static Pod由Kubelet来控制管理。Kubernetes API Server可以在每个Kubernetes Master节点上运行，前端通过负载均衡器作为访问入口。

另外，每个Kubernetes Master节点上需要运行一个小程序Podmaster，Podmaster通过Etcd进行选举，从多个Kubernetes Master中选择主节点，只有主节点上会运行Kubernetes Scheduler和Kubernetes Controller Manager。当主节点发生宕机时，选择新的主节点运行Kubernetes Scheduler和Kubernetes Controller Manager。

12.3 平台监控

监控是整个运维环节，乃至整个产品生命周期中最重要的一环，事前及时预警发现故障，事后提供翔实的数据用于追查定位问题。对于Kubernetes来说，监控的层级和需求是多样的，对于系统管理员来说，希望Kubernetes系统级别的监控，比如Node的资源消耗数据用来判断是否需要增加新的机器。而对于使用人员来说，希望应用本身监控，包括Pod或者容器的详细运行数据，用以了解应用的性能情况和瓶颈所在。

Kubernetes提供了平台监控支持，安装方法可参考2.3.2节，收集Kubernetes各个维度的数据，Kubernetes平台监控的逻辑设计如图12-2所示。

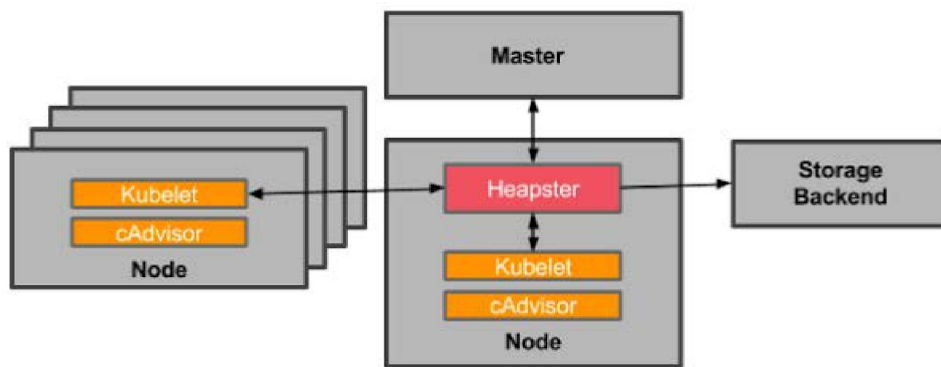


图12-2 Kubernetes平台监控

Kubernetes平台监控的两个关键组件是cAdvisor和Heapster。

12.3.1 cAdvisor

cAdvisor (Container Advisor) 是谷歌开源的一个容器监控工具。它是一个守护进程，收集、统计和处理宿主机和容器的数据，包括实时和历史的CPU、内存、磁盘、网络使用情况。同时，cAdvisor提供良好的Web界面和Rest API来展示数据，帮助系统管理者清楚了解容器的资源使用情况和运行性能数据。

提示

在cAdvisor的认识中，容器的概念是广义的，不仅仅包括Docker容器，也包括其他容器实现，另外宿主机本身也是一种容器，称为根容器，它是一种原生容器。

Web UI

cAdvisor提供了一个Web界面可以查询监控数据，在cAdvisor运行后，便可以访问cAdvisor的Web界面，如图12-3所示。

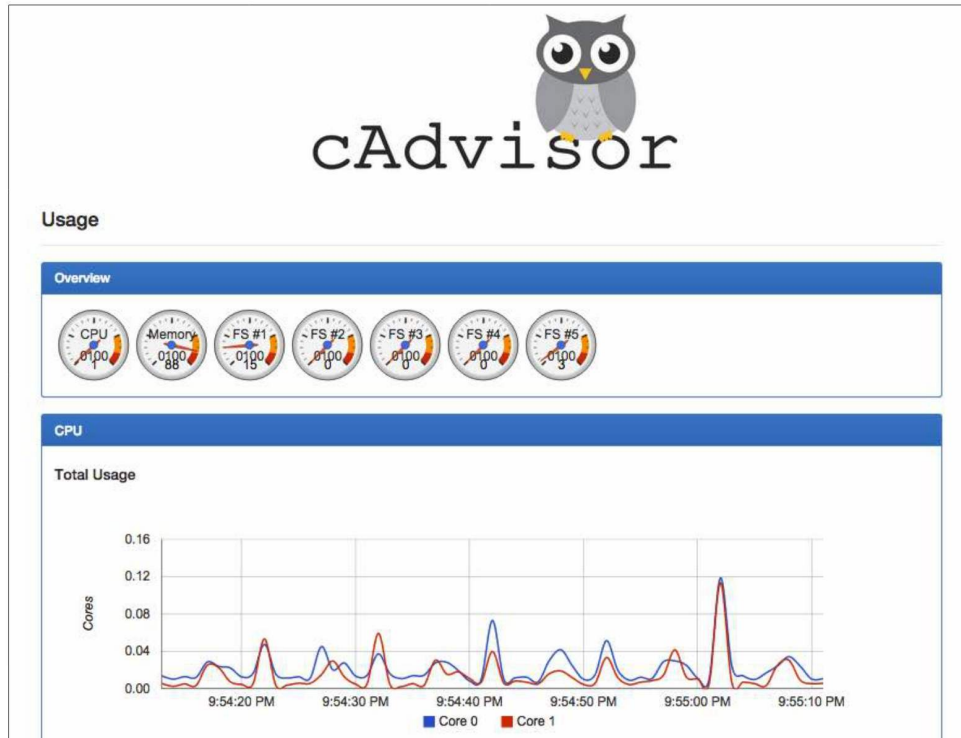


图12-3 cAdvisor Web界面

REST API

cAdvisor提供REST API用于获取监控数据，cAdvisor的REST API访问方式为`http:// <hostname>:<port>/api/<version>/<request>`。

cAdvisor REST API支持的版本和请求类型如下所示。

- v1.0: containers,machine
- v1.1: containers,machine,subcontainers
- v1.2: containers,docker,machine,subcontainers
- v1.3: containers,docker,events,machine,subcontainers

• v2.0 :
appmetrics,attributes,events,machine,ps,spec,stats,storage,summary,version

其中v1.3是最新的稳定版本， v2.0处于beta阶段， 具体说明如表12-1所示。

表12-1 cAdvisor REST API

版本	URL	说明
V1	GET /api/v1.x/machine	获取宿主机信息
	GET /api/v1.x/containers/<container>	获取指定容器的监控数据
	GET /api/v1.x/subcontainers/<container>	获取指定子容器的监控数据
	GET /api/v1.x/docker	获取所有Docker容器的监控
	GET /api/v1.x/docker/<container>	获取指定Docker容器的监控数据。可以使用Docker容器UUID或者名称
	GET /api/v1.x/events/<container>	获取指定容器的事件信息
V2	GET /api/v2.X/version	获取cAdvisor的版本
	GET /api/v2.x/machine	获取宿主机信息
	GET /api/v2.x/attributes	获取宿主机的硬件和软件属性
	GET /api/v2.x/storage	获取宿主机的存储设备信息
	GET /api/v2.x/stats/<container>	获取指定容器的统计数据。Docker容器需要增加参数type=docker
	GET /api/v2.x/summary/<container>	获取指定容器的统计汇总数据。Docker容器需要增加参数type=docker
	GET /api/v2.x/spec/<container>	获取指定容器的规格。Docker容器需要增加参数type=docker
	GET /api/v2.x/ps/<container>	获取指定容器包含的所有进程信息。Docker容器需要增加参数type=docker
	GET /api/v2.x/appmetrics/<container>	支持自定义应用的监控维度， 通过API获取监控数据

Kubelet集成

Kubelet组件已经集成了cAdvisor， 在Kubernetes Node上可以直接访问cAdvisor， cAdvisor运行端口可以通过Kubelet的启动参数 --cadvisor-port设置， 默认是4194。

Kubelet基于cAdvisor提供了REST API（默认端口10255）来获取Node和Pod/容器的监控数据。

- 获取Node的监控统计数据，如表12-2所示。

表12-2 获取Node的监控统计数据

URL	GET /stats/
参数	num_stats: 返回最大的统计数据条数，默认值为60 start: 统计的起始时间，默认最开始的时间 end: 统计的结束时间，默认当前时间

- 获取非Kubernetes容器的监控统计数据，如表12-3所示。

表12-3 获取非Kubernetes容器的监控统计数据

URL	GET /stats/container
参数	containerName: 需要统计的容器名称，默认值为“/” subcontainers: 是否包含子容器，默认值为false num_stats: 返回最大的统计数据条数，默认值为60 start: 统计的起始时间，默认最开始的时间 end: 统计的结束时间，默认当前时间

- 获取Kubernetes容器的监控统计数据，如表12-4所示。

表12-4 获取Kubernetes容器的监控统计数据

URL	GET /stats/<namespace>/<pod_name>/<container_name> GET /stats/<pod_name>/<container_name> (查询默认Namespace)
参数	num_stats: 返回最大的统计数据条数, 默认值为60 start: 统计的起始时间, 默认最开始的时间 end: 统计的结束时间, 默认当前时间

Kubelet统计API返回的监控统计数据的格式如下所示。

- name: 容器名称。
- aliases: 容器别名。
- namespace: 命名空间, 由cAdvisor定义。
- subcontainers: 包含的子容器列表。
- spec: 容器规格, 包含如下信息。
 - creation_time: 创建时间。
 - labels: 包含的标签。
 - has_cpu: 是否有CPU。
 - cpu: CPU信息。
 - has_memory: 是否有内存。
 - memory: 内存信息。
 - has_network: 是否有网络。

- `has_filesystem`: 是否有文件系统。
- `has_diskio`: 是否有磁盘IO。
- `image`: 容器镜像。
- `stats`: 容器监控统计数据，每条数据包含如下信息。
 - `timestamp`: 时间戳。
 - `cpu`: CPU统计数据。
 - `diskio`: 磁盘IO统计数据。
 - `memory`: 内存统计数据。
 - `network`: 网络统计数据。
 - `filesystem`: 文件系统统计数据。
- `task_stats`: 任务统计数据。

12.3.2 Heapster

Heapster是谷歌开源的容器集群的监控收集工具，它可以集成Kubernetes进行监控数据的收集汇总，提供REST API来获取Kubernetes各个维度的监控数据。另外，Heapster支持对接第三方系统，将监控数据导入到第三方系统进行进一步处理。

REST API

Heapster提供REST API，如表12-5所示。

表12-5 Heapster REST API

类型	URL	说明
metric	GET /api/v1/metric-export	获取最新的Metric数据
	GET /api/v1/metric-export-schema	获取Metric数据的规格
Sink	POST /api/v1/sinks	配置当前的Sink
	GET /api/v1/sinks	获取当前的Sink

集成Kubernetes

Heapster在与Kubernetes集成的时候，Heapster调用Kubernetes API 获取所有Node列表，然后调用Kubelet的API收集汇总监控数据。同时，Heapster根据收集到的数据建立一个Kubernetes监控模型（Metric Model），包括Kubernetes以下层级。

- Cluster: 整个Kubernetes运行环境的监控模型。
- Node: 各个Node的监控数据模型，包括机器本身的监控和Node上运行的Pod的监控。
- Namespace: 各个Namespace监控模型，相当于Namespace下所有Pod的监控。
- Pod: 各个Pod的监控模型。
- Container: 各个Container监控模型。

Kubernetes监控模型提供API来获取各个层级的监控数据，如表12-6所示。

表12-6 Kubernetes 监控模型

层级	URL
Cluster	/api/v1/model/
	/api/v1/model/metrics/
	/api/v1/model/metrics/<metric-name>
	/api/v1/model/stats/
Node	/api/v1/model/nodes/
	/api/v1/model/nodes/<node-name>/
	/api/v1/model/nodes/<node-name>/pods/
	/api/v1/model/nodes/<node-name>/metrics/
	/api/v1/model/nodes/<node-name>/metrics/<metric-name>
	/api/v1/model/nodes/<node-name>/stats/

续表

层级	URL
Namespace	/api/v1/model/namespaces/
	/api/v1/model/namespaces/<namespace-name>/
	/api/v1/model/namespaces/<namespace-name>/metrics/
	/api/v1/model/namespaces/<namespace-name>/metrics/<metric-name>
	/api/v1/model/namespaces/<namespace-name>/stats/
Pod	/api/v1/model/namespaces/<namespace-name>/pods/
	/api/v1/model/namespaces/<namespace-name>/pods/{pod-name}/
	/api/v1/model/namespaces/<namespace-name>/pods/{pod-name}/metrics/
	/api/v1/model/namespaces/<namespace-name>/pods/<pod-name>/metrics/<metric-name>
	/api/v1/model/namespaces/<namespace-name>/pods/<pod-name>/stats/
Container	/api/v1/model/namespaces/<namespace-name>/pods/<pod-name>/containers/
	/api/v1/model/namespaces/<namespace-name>/pods/<pod-name>/containers/<container-name>/
	/api/v1/model/namespaces/{namespace-name}/pods/<pod-name>/containers/<container-name>/metrics/
	/api/v1/model/namespaces/<namespace-name>/pods/<pod-name>/containers/<container-name>/metrics/<metric-name>
	/api/v1/model/namespaces/<namespace-name>/pods/<pod-name>/containers/<container-name>/stats/
	/api/v1/model/nodes/<node-name>/freecontainers/
	/api/v1/model/nodes/<node-name>/freecontainers/<container-name>/
	/api/v1/model/nodes/<node-name>/freecontainers/<container-name>/metrics/
	/api/v1/model/nodes/{node-name}/freecontainers/{container-name}/metrics/{metric-name}
	/api/v1/model/nodes/{node-name}/freecontainers/{container-name}/stats/

12.4 平台日志

Kubernetes提供了平台日志支持，安装方法可参考2.3.3节。平台日志基于Fluentd+ Elasticsearch+Kibana，其中Fluentd作为Logging

Agent收集日志汇总到Elasticsearch。

Fluentd是一个开源的日志收集系统，支持150多个插件，能够将日志收集到MongoDB、Redis、Amazon S3和Elasticsearch等；Fluentd能够以JSON格式处理日志，具备每天收集5000+台服务器上5TB的日志数据，每秒处理50000条消息的性能。

Fluentd运行在Kubernetes的所有节点上，收集Kubernetes组件的日志，Fluentd的配置如下所示：

```
<source>
  type tail
  format none
  path /var/log/docker.log
  pos_file /var/log/es-docker.log.pos
  tag docker
</source>
```

```
<source>
  type tail
  format none
  path /var/log/etcd.log
  pos_file /var/log/es-etcd.log.pos
  tag etcd
</source>
```

```
<source>
  type tail
```



```
format none
path /var/log/kubelet.log
pos_file /var/log/es-kubelet.log.pos
tag kubelet
</source>
```

```
<source>
  type tail
  format none
  path /var/log/kube-apiserver.log
  pos_file /var/log/es-kube-apiserver.log.pos
  tag kube-apiserver
</source>
```

```
<source>
  type tail
  format none
  path /var/log/kube-controller-manager.log
  pos_file /var/log/es-kube-controller-manager.log.pos
  tag kube-controller-manager
</source>
```

```
<source>
  type tail
  format none
  path /var/log/kube-scheduler.log
  pos_file /var/log/es-kube-scheduler.log.pos
```

```
    tag kube-scheduler
</source>
```

Fluentd配置中的Source用于指定日志输入资源，其中字段含义如下所示。

- **type**: 指定日志的输入方式，其中**tail**方式是不停地从源文件中获取新的日志。
- **format**: 指定日志的输出格式。
- **path**: 指定日志文件的位置。
- **tag**: 指定日志tag，用来对不同的日志进行分类。

可以看出，Fluentd将监控每个组件的日志文件，然后设置上tag，最后输出。另外，Fluentd同时会监控容器的日志文件：

```
<source>
  type tail
  path /var/log/containers/*.log
  pos_file /var/log/es-containers.log.pos
  time_format %Y-%m-%dT%H:%M:%S
  tag kubernetes.*
  format json
  read_from_head true
</source>
```

Fluentd会监控/var/log/containers目录下的所有日志文件，然后以JSON格式输出，其中设置的tag是kubernetes.*，在tail方式下会被设置

为kubernetes.path.to.file。

实际上，/var/log/containers目录是由Kubelet组件创建的，Kubelet在其中建立Pod的容器日志文件，这些是容器中应用打印到标准输出（Stdout）的日志，即通过kubectl logs或者docker logs查询到的日志。

比如我们运行的Hello World Pod，其信息查询如下：

```
$ kubectl describe pod hello-world
```

```
Name: hello-world
```

```
Namespace: default
```

```
Image(s): ubuntu:14.04
```

```
Node: kube-node-2/192.168.3.148
```

```
Start Time: Fri, 04 Dec 2015 19:28:21 +0800
```

```
Labels: <none>
```

```
Status: Succeeded
```

```
Reason:
```

```
Message:
```

```
IP:
```

```
Replication Controllers:<none>
```

```
Containers:
```

```
hello:
```

```
Container ID:
```

```
docker://8c9df43b96227d14f8665d87d5b32ee39ce11328bd1d1848465c3c  
9e97a09b8a
```

```
Image: ubuntu:14.04
```

```
Image ID:
```

```
docker://8251da35e7a79dca688682f6da6148a06d358c6f094020844468a7
82842c2172
```

```
State: Terminated
```

```
Reason: Error
```

```
Exit Code: 0
```

```
Started: Fri, 04 Dec 2015 19:28:28 +0800
```

```
Finished: Fri, 04 Dec 2015 19:28:29 +0800
```

```
Ready: False
```

```
Restart Count: 0
```

```
Environment Variables:
```

```
.....
```

Hello World Pod包含的一个容器hello将输出Hello World，这条输出日志实际上是由Docker进行存储，并通过Kubernetes获取的，而Fluentd将会进行日志收集。

```
$ kubectl logs hello-world hello
```

```
Hello World
```

Docker容器的日志都会由Docker进行存储，而Kubelet会在/var/log/containers下生成一个文件软连接Docker存储的容器日志文件，文件格式是 [pod_name]_[namespace]_ [container_name]-[container_id].log:

```
$ cat hello-world_default_hello-8c9df43b96227d14f8665d87d5b32ee39ce11328bd1d1848465-c3c9e97a09b8a.log
```

```
{"log": "Hello      World\n", "stream": "stdout", "time": "2015-12-04T11:28:29.016916036Z"}
```

那么Fluentd将读取这个日志文件，对应的tag是：

```
kubernetes.var.log.containers.hello-world_default_hello-8c9df43b96227d14f8665d87d5b32ee39ce11328bd1d1848465c3c9e97a09b8a.log
```

Fluentd最终将日志导入到Elasticsearch，通过检索Elasticsearch可以查询到该日志：

```
{
  "_index": "logstash-2015.12.04",
  "_type": "fluentd",
  "_id": "AVFs0tILXx_4Q0Dgz43x",
  "_score": 6.7633038,
  "_source": {
    "log": "Hello World\n",
    "stream": "stdout",
    "docker": {
      "container_id":
"8c9df43b96227d14f8665d87d5b32ee39ce11328bd1d1848465c3c9e97a09b8a"
    },
    "kubernetes": {
      "namespace": "default",
      "pod_name": "hello-world",
```

```
        "container_name": "hello"
    },
    "tag":
"kubernetes.var.log.containers.hello-world_default_hello-
8c9df43b96227d14f8665d87d5b32ee3
9ce11328bd1d1848465c3c9e97a09b8a.log",
    "@timestamp": "2015-12-04T11:28:29+00:00"
}
}
```

12.5 垃圾清理

Kubernetes系统在长时间运行后，Kubernetes Node会下载非常多的镜像，其中可能会存在很多过期的镜像。同时因为运行大量的容器，容器退出后就变成死亡容器，将数据残留在宿主机上，这样一来，过期镜像和死亡容器都会占用大量的硬盘空间。如果硬盘空间被用光，可能会发生非常糟糕的情况，甚至会导致硬盘的损坏。为此，Kubelet会进行垃圾清理工作，即定期清理过期镜像和死亡容器。

12.5.1 镜像清理

镜像清理的策略是当硬盘空间使用率超过阈值的时候开始执行，Kubelet执行清理的时候优先清理最久没有被使用的镜像。

硬盘空间使用率的阈值通过Kubelet的启动参数--image-gc-high-threshold和--image-gc- low-threshold指定。

12.5.2 容器清理

Kubelet容器清理的相关参数如表12-7所示。

表12-7 Kubelet容器清理参数

参数	Kubelet启动参数	说明
MinAge	--minimum-container-ttl-duration	死亡容器能够被删除的最小TTL，默认是1分钟
MaxPerPodContainer	--maximum-dead-containers-per-container	每个Pod允许存在的最大死亡容器数目，默认是2
MaxContainers	--maximum-dead-containers	允许存在的最大死亡容器数目，默认是100

Kubelet定时执行容器清理，每次根据以上3个参数选择死亡容器删除，通常情况下优先删除创建时间最久的死亡容器。Kubelet不会删除非Kubelet管理的容器。

12.6 Kubernetes的Web界面

Kubernetes 提供了一个 Web 界面 Kube UI (<https://github.com/kubernetes/kube-ui>)，用来图形化展示运行环境信息。安装方法可参考2.3.4节。安装成功后可以通过Kubernetes API Server的接口`http://<kubernetes-master>/ui`进行访问。

Kube UI首页图形化展示了所有Kubernetes Node的资源使用情况，包括CPU、Memory和Filesystem的使用情况，如图12-4所示。



图12-4 Kube UI首页

单击首页右上角的View按钮，包含Explore、Pods、Nodes、Replication Controllers、Services和Events等选项。单击进入Explore页面，会列出所有Pod、Replication Controller和Service，支持筛选和排列展示，如图12-5所示。

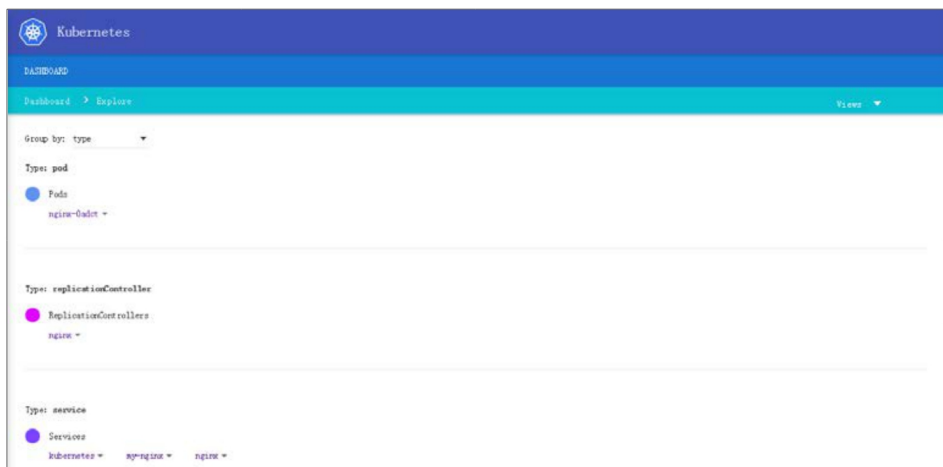


图12-5 Explore页面

同时可以分别查询Pod、Replication Controller和Service的详细信息，比如查询Service的详细信息，如图12-6所示。

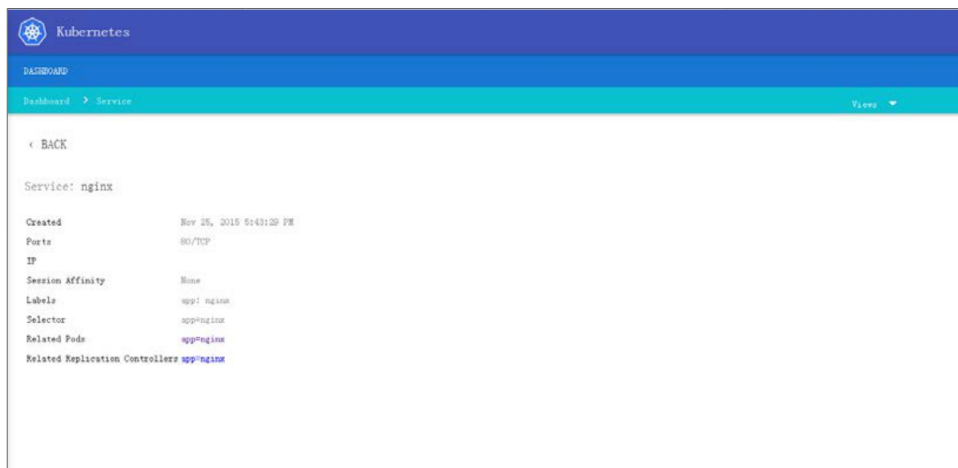


图12-6 查询Service的详细信息

其他页面也列出了相应的信息，比如查询Event，如图12-7所示。

First Seen	Last Seen	Count	Reason	End	SubObject	Reason	Source	Message
Nov 24, 2015 10:21:29 PM	Nov 25, 2015 6:58:01 PM	18	kube-node-2	Node		NodeNotReady	controllermanager undefined	Node kube-node-2 status is now: NodeNotReady
Nov 25, 2015 6:08:29 PM	Nov 25, 2015 6:58:02 PM	4	kube-node-2	Node		NodeReady	kubelet kube-node-2	Node kube-node-2 status is now: NodeReady
Nov 24, 2015 7:46:22 PM	Nov 25, 2015 7:15:36 PM	140	kube-node-3	Node		NodeReady	kubelet kube-node-3	Node kube-node-3 status is now: NodeReady
Nov 24, 2015 8:53:34 PM	Nov 25, 2015 7:15:36 PM	128	kube-node-3	Node		NodeNotReady	controllermanager undefined	Node kube-node-3 status is now: NodeNotReady
Nov 25, 2015 5:44:36 PM	Nov 25, 2015 7:22:53 PM	177	nginx	HorizontalPodAutoscaler		FailedGetMetrics	horizontal-pod-autoscaler undefined	failed to get CPU consumption and request: metrics obtained for 0/1 of pods
Nov 25, 2015 5:44:36 PM	Nov 25, 2015 7:22:53 PM	177	nginx	HorizontalPodAutoscaler		FailedComputeReplicas	horizontal-pod-autoscaler undefined	failed to get cpu utilization: failed to get CPU consumption and request: metrics obtained for 0/1 of pods

图12-7 查询Event

Kube UI 只能简单地查看一些信息，不能进行修改，新的 Kubernetes Dashboard (<https://github.com/kubernetes/dashboard>) 正在开发中，将支持功能更加强大的Web界面。

第3部分

Kubernetes生态篇

第13章 CoreOS

第14章 Etcd

第15章 Mesos

第13章

CoreOS

CoreOS是容器生态圈重要的一环，与Kubernetes和Docker都有着非常密切的关系。本章将简单介绍CoreOS，然后讲解如何安装CoreOS，以及如何使用CoreOS运行Kubernetes。

13.1 CoreOS介绍

CoreOS是一个轻量级、容器化的Linux发行版，借助了以Docker为代表的容器技术，专为大型数据中心而设计，旨在通过轻量的系统架构和灵活的应用程序部署能力简化数据中心的维护成本和复杂度。

CoreOS没有提供包管理工具，是通过容器化的运算环境向应用程序提供运算资源的。在CoreOS中，所有应用程序都被装在容器中运行在操作系统之上，可以很轻松地将应用程序在操作系统之间转移。

应用程序之间共享系统内核和资源，但是彼此之间又互不可见。这意味着应用程序将不会再被直接安装到操作系统中，而是运行在容器中。这种方式使得操作系统、应用程序及运行环境之间的耦合度大大降低。相对于传统的部署方式而言，在CoreOS集群中部署应用程序更加灵活便捷，应用程序运行环境之间的干扰更少，而且操作系统自身的维护也更加容易。

借助容器的能力，CoreOS可以剔除任何不必要的软件和服务，最小化定制化Linux系统。因此很小很轻量，管理员操心的事情会少很多，允许快速修复，占据的空间也很小。

在一定程度上减轻了维护一个服务器集群的复杂度，可帮助用户从烦琐的系统及软件维护工作中解脱出来。

13.2 CoreOS工具链

13.2.1 Etcd

在CoreOS集群中处于骨架地位的是Etcd，Etcd是Core团队开发的一个高可用的键值存储系统，灵感来自于ZooKeeper和Doozer。Etcd以默认的形式安装于每个CoreOS系统之中，CoreOS集群中的程序和服务可以通过Etcd共享信息或服务发现。

13.2.2 Flannel

Flannel是CoreOS团队开发的覆盖网络工具，CoreOS集群使用Flannel创建的一个容器覆盖网络，实现容器之间的通信报文，实现跨主机通信。

13.2.3 Rocket

Rocket是CoreOS推出的一款容器引擎，和**Docker**类似，帮助开发者打包应用和依赖包到可移植容器中，简化搭建环境等部署工作。**Rocket**同**Docker**相比更加专注于容器核心技术和容器标准。

13.2.4 Systemd

Systemd是Linux下的一种Init软件，由Lennart Poettering带头开发，并在LGPL 2.1及其后续版本许可证下开源发布框架以表示系统服务间的依赖关系，并依此实现系统初始化时服务的并行启动，同时达到降低Shell的系统开销的效果，最终代替现在常用的**System V**与**BSD**风格的Init程序，CoreOS已将**Systemd**作为Linux发行版的Init系统。

13.2.5 Fleet

Fleet是一个通过**Systemd**对CoreOS集群进行控制和管理工具。**Fleet**与**Systemd**之间通过D-Bus API进行交互，每个**FleetAgent**之间通过Etcd服务来注册和同步数据。**Fleet**提供的功能非常丰富，包括查看集群中服务器的状态、启动或终止**Docker**容器、读取日志内容等。更为重要的是，**Fleet**可以确保集群中的服务一直处于可用状态。当出现某个通过**Fleet**创建的服务在集群中不可用时，如由于某台主机因为硬件或网络故障从集群中脱离时，原本运行在这台服务器中的一系列服务将通过**Fleet**被重新分配到其他可用服务器中。虽然当前**Fleet**还处于初期状态，但是其管理CoreOS集群的能力是非常有效的，并且仍然有很大的扩展空间，目前已提供简单的API接口供用户集成。

13.3 CoreOS实践

13.3.1 安装CoreOS

本节将使用Vagrant部署CoreOS，这可以说是最简单的方式了，可以使用本地虚拟机快速部署CoreOS。

准备工作

在机器上预先装好VirtualBox和Vagrant:

- VirtualBox 4.3.10以上版本
- Vagrant 1.6以上版本

下载CoreOS-Vagrant仓库，其中包含使用Vagrant部署CoreOS的配置文件:

```
$ git clone https://github.com/coreos/coreos-vagrant
$ cd coreos-vagrant
```

配置

在CoreOS-Vagrant仓库目录下，config.rb.sample和user-data.sample是两个模板文件，根据这两个模板文件可以对CoreOS部署进行自定义配置，通过模板文件生成配置文件:

```
$ cp config.rb.sample config.rb
$ cp user-data.sample user-data
```


配置文件user-data将作为CoreOS操作系统初始化时使用的Cloud-Init配置文件，Cloud-Init是专为云环境中虚拟机而开发的工具，它根据配置文件对虚拟机进行系统初始化配置，配置文件使用YAML文件格式，并且需要包含#cloud-config:

```
#cloud-config
coreos:
  etcd2:
    #generate a new token for each unique cluster from
    https://discovery.etcd.io/new
    #discovery: https://discovery.etcd.io/<token>
    # multi-region and multi-cloud deployments need to use
    $public_ipv4
    advertise-client-urls: http://$public_ipv4:2379
    initial-advertise-peer-urls: http://$private_ipv4:2380
    # listen on both the official ports and the legacy ports
    # legacy ports can be omitted if your application doesn't
    depend on them
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
                                listen-peer-urls:
    http://$private_ipv4:2380,http://$private_ipv4:7001
  fleet:
    public-ip: $public_ipv4
  flannel:
    interface: $public_ipv4
  units:
    - name: etcd2.service
```

```

    command: start
- name: fleet.service
    command: start
- name: flanneld.service
    drop-ins:
      - name: 50-network-config.conf
        content: |
            [Service]
                                ExecStartPre=/usr/bin/etcdctl set
/coreos.com/network/config '{ "Network":
"10.1.0.0/16" }'
    command: start
- name: docker-tcp.socket
    command: start
    enable: true
    content: |
        [Unit]
        Description=Docker Socket for the API

        [Socket]
        ListenStream=2375
        Service=docker.service
        BindIPv6Only=both
        [Install]
        WantedBy=sockets.target

```

Cloud-Init配置文件中包含了Etcd、Fleet、Flannel和Docker的服务启动参数，其中使用两个变量\$private_ipv4和\$public_ipv4，它们会在实际运行的时候被自动替换为虚拟机的真实外网IP和内网IP地址。用户可以根据需要，在配置中添加更多定制化的服务和配置，具体信息可参考 <https://coreos.com/docs/cluster-management/setup/cloudinit-cloud-config>。

另外，CoreOS集群是用Etcd进行服务发现的，Cloud-Init配置文件中的.coreos.etcd2.discovery就是用来配置一个外部Etcd服务的URL，CoreOS启动时通过这个集群标识URL地址自动进入同一个集群中，这就实现了无须人工干预的集群服务器自发现。我们可以通过Etcd发现服务（discovery.etcd.io）申请创建URL，如果没有配置的话，在config.rb中也会自动申请创建进行配置。

配置文件config.rb中包含了Vagrant虚拟机的配置，通过这个文件可以覆盖Vagrantfile里的参数。我们主要关注\$num_instances和\$update_channel这两个参数：

- \$num_instances表示将启动的CoreOS集群中需要包含主机实例的数量。
- \$update_channel表示启动的CoreOS实例使用的升级通道，可以是stable、eta或alpha。

现在配置config.rb，将部署有1个节点的CoreOS集群，并使用alpha升级通道：

```
$num_instances=1
$update_channel='alpha'
```

...

部署

使用Vagrant部署CoreOS:

```
$ vagrant up
```

部署成功后，可以查询虚拟机的状态:

```
$ vagrant status
```

Current machine states:

```
core-01                running (virtualbox)
```

```
core-02                running (virtualbox)
```

```
core-03                running (virtualbox)
```

This environment represents multiple VMs. The VMs are all listed above with their current state. For more information about a specific

VM, run `vagrant status NAME`.

在 CoreOS 集群运行后，集群的实例之间通过 Etcd 实现自发现服务，同时运行起Docker和Flannel，形成连通的容器集群，可以SSH到CoreOS节点中:

```
$ vagrant ssh core-01
```

然后在CoreOS节点中查询服务状态:

```
$ sudo systemctl status etcd2 docker flanneld
```

13.3.2 使用CoreOS运行Kubernetes

Kubernetes是一个容器集群管理平台，而CoreOS是基于容器的集群操作系统，两者可以说有着千丝万缕的关系。CoreOS集群已经原生运行Docker，同时使用Flannel搭建出容器的覆盖网络，因此Kubernetes非常适合运行在CoreOS之上。

现在运行3个节点的CoreOS集群，我们将在CoreOS集群之上运行Kubernetes，其中所有节点将作为Kubernetes Node，而节点core-01同时作为Kubernetes Master。

启动Kubelet

CoreOS在alpha开发版（773.1.0以后版本）中集成了Kubernetes的核心组件Kubelet，我们通过在所有节点中进行简单配置就可以启动kubelet，从而作为Kubernetes Node。

创建kubelet Systemd单元文件/etc/systemd/system/kubelet.service:

```
[Unit]
Description=Kubernetes Kubelet
Documentation=https://github.com/kubernetes/kubernetes

[Service]
ExecStartPre=/usr/bin/mkdir -p /etc/kubernetes/manifests
ExecStart=/usr/bin/kubelet \
--api-servers=http://kuber-apiserver:8080 \
--allow-privileged=true \
```

```
--config=/etc/kubernetes/manifests \
```

```
--v=2
```

```
Restart=on-failure
```

```
RestartSec=5
```

```
[Install]
```

```
WantedBy=multi-user.target :qw
```

其中，我们将在节点 core-01 上运行 Kubernetes API Server，所以--api-servers 设置的URL指向core-01。

配置好Systemd单元文件后，使用systemctl命令启动Kubelet:

```
$ sudo systemctl daemon-reload
```

```
$ sudo systemctl start kubelet
```

运行后用systemctl命令查询Kubelet是否运行:

```
$ sudo systemctl status kubelet
```

另外设置kubelet为开机自启动:

```
$ sudo systemctl enable kubelet
```

部署Kubernetes Master

现在在节点core-01上运行Kubernetes Master，首先下载Kubernetes Master的Pod定义文件:

```
$ wget  
https://raw.githubusercontent.com/coreos/pods/master/kubernetes.  
yaml
```

然后将其放入Kubelet的Manifest目录，由Kubelet以Daemon Pod的形式运行Kubernetes Master各组件：

```
$ sudo cp kubernetes.yaml /etc/kubernetes/manifests/
```

确保镜像下载正常，那么一段时间后，Kubernetes Master的各个组件将运行起来。

第14章

Etcd

Kubernetes 使用 Etcd 进行存储，理解和掌握 Etcd 对于管理 Kubernetes 至关重要。本章将介绍 Etcd，包括基本概念和结构组织，然后详细说明如何运行和管理 Etcd。

14.1 Etcd介绍

Etcd是一个高可用的键值存储系统，Etcd的灵感来自于ZooKeeper和Doozer，通过Raft共识算法（The Raft Consensus Algorithm）处理日志复制以保证强一致性。

Etcd中的主要概念如下所示。

- Raft: Etcd所采用的保证分布式系统强一致性的算法。
- Node: 一个Raft状态机实例。
- Member: 一个Etcd实例，它管理着一个Node，并且可以为客户端请求提供服务。
- Cluster: 由多个Member构成可以协同工作的Etcd集群。
- Peer: 对同一个Etcd集群中另外一个Member的称呼。
- Client: 向Etcd集群发送HTTP请求的客户端。

- **WAL**: 预写式日志, **Etcd**用于持久化存储的日志格式。
- **Snapshot**: **Etcd**防止**WAL**文件过多而设置的快照, 存储**Etcd**数据状态。
- **Proxy**: **Etcd**的一种模式, 为**Etcd**集群提供反向代理服务。
- **Leader**: **Raft**算法中通过竞选而产生的处理所有数据提交的节点。
- **Follower**: 竞选失败的节点作为**Raft**中的从属节点, 为算法提供强一致性保证。
- **Candidate**: 当**Follower**超过一定时间接收不到**Leader**的心跳时转变为**Candidate**开始竞选。
- **Term**: 某个节点成为**Leader**到下一次竞选的时间, 称为一个**Term**。
- **Index**: 数据项编号。Raft中通过**Term**和**Index**来定位数据。

14.2 Etcd的结构

Etcd主要分为4个部分, 如图14-1所示。

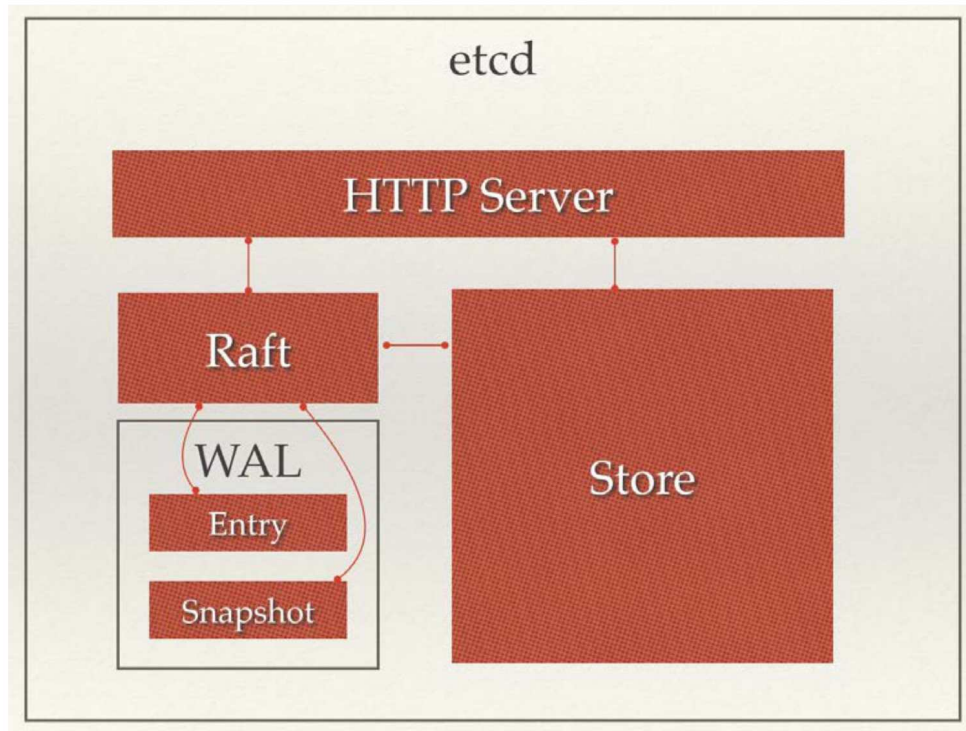


图14-1 Etcd的结构

- **HTTP Server**: 用于处理用户发送的API请求以及其他Etcd节点的同步与心跳信息请求。

- **Store**: 用于处理Etcd支持的各类功能的事务，包括数据索引、节点状态变更、监控与反馈、事件处理与执行等，是Etcd对用户提供的绝大多数API功能的具体实现。

- **Raft**: 强一致性算法的具体实现，是Etcd的核心。

- **WAL**: Write Ahead Log（预写式日志），是Etcd的数据存储方式，是用于向系统提供原子性和持久性的一系列技术。在使用WAL的时候，所有的修改在提交之前都要先写入日志文件中。Etcd的WAL由日志存储与快照存储两部分组成，其中，**Entry**负责存储具体日志的内

容，而Snapshot负责在日志内容发生变化的时候保存Raft的状态。WAL会在本地磁盘的一个指定目录下分别存放日志条目与快照内容。

通常，一个用户的请求发送过来，会经由HTTP Server转发给Store进行具体的事务处理。如果涉及节点的修改，则交给Raft模块进行状态的变更、日志的记录，然后再同步给其他Etcd节点以确认数据提交，最后进行数据的提交、再次同步。

Etcd属于分布式架构，其中的通信模型有两种，一种是Etcd Client同Etcd Server之间的通信，另一种是Etcd Peer之间的通信。

14.2.1 Client-to-Server

在默认设置下，Etcd通过主机的2379/4001端口向Client提供服务，每个主机上的应用程序都可以通过主机的2379/4001端口以HTTP + JSON的方式向Etcd读写数据。写入的数据会由Etcd同步到集群的其他节点中，如图14-2所示。

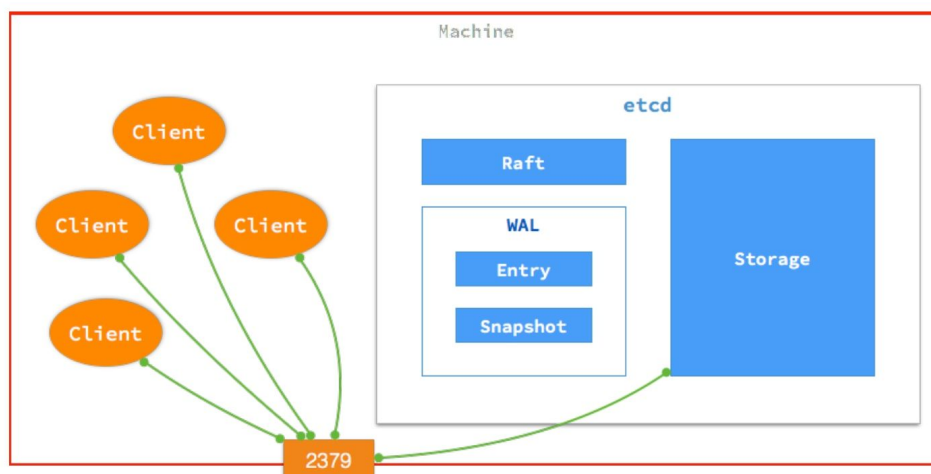


图14-2 Etcd Client同Etcd Server之间的通信

14.2.2 Peer-to-Peer

在默认设置下，Etcd通过主机的2380/7001端口在各个节点中同步Raft状态及数据，如图14-3所示。

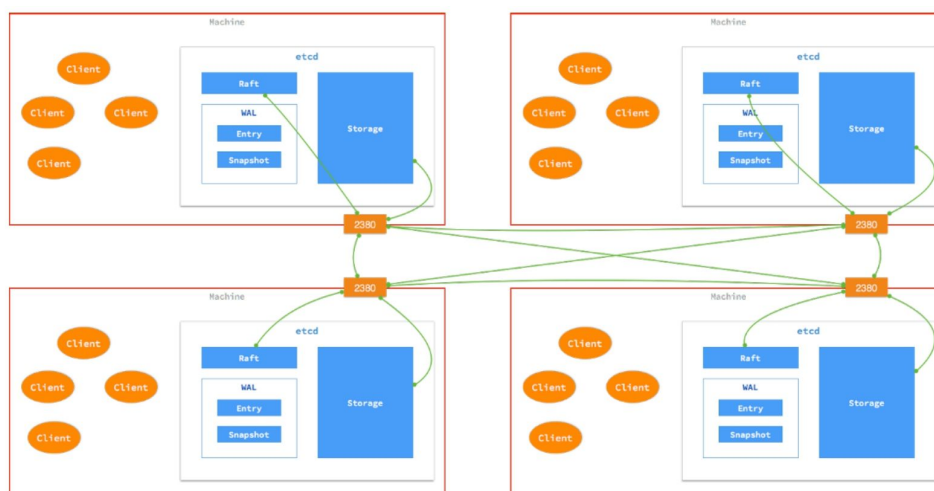


图14-3 Etcd Peer之间的通信

14.3 Etcd实践

14.3.1 运行Etcd

可以从Github上下载指定版本的Etcd发布包进行安装：

```
$ wget https://github.com/coreos/etcd/releases/download/v2.2.0/etcd-v2.2.0-linux-amd64.tar.gz
$ tar xzvf etcd-v2.2.0-linux-amd64.tar.gz
```

```
$ cd etcd-v2.2.0-linux-amd64
```

```
$ cp ./etcd /usr/bin
```

启动单节点的Etcd:

```
$ etcd -name etcd \
```

```
-data-dir /var/lib/etcd \
```

```
-listen-client-urls http://0.0.0.0:2379,http://0.0.0.0:4001 \
```

```
-advertise-client-urls http://0.0.0.0:2379,http://0.0.0.0:4001
```

在Etcd的启动参数中，`-name`设置Etcd节点的名称，`-data-dir`设置Etcd的数据目录。另外，`-listen-client-urls`设置了Etcd向Client提供服务的监听URL，多个URL直接用逗号分隔，而`-advertise-client-urls`指定了Etcd对外广播的URL，需要在`-listen-client-urls`配置URL范围。

14.3.2 Etcd集群化

Etcd集群的工作原理基于Raft共识算法。Raft共识算法的优点在于，可以在高效地解决分布式系统中各个节点日志内容一致性问题的同时，也使得集群具备一定的容错能力，即使集群中出现部分节点故障、网络故障等问题，仍可保证其余大多数节点正确地步进，甚至当更多的节点（一般来说超过集群节点总数的一半）出现故障而导致集群不可用时，依然可以保证节点中的数据不会出现错误的结果。

因为需要选举Leader节点，所以Etcd集群至少需要两个节点，但是如果是两个成员，那么在一台无法正常运转的情况下，剩下的一个

也无法正常工作，一般建议是3~9个节点，并且一个重要的集群优化策略是要保障集群中活跃节点的数目始终为奇数个。

14.3.2.1 搭建集群

要搭建Etcd集群，需要让Etcd节点互相发现，目前提供的发现方式有3种：

- 静态发现
- Etcd动态发现
- DNS服务发现

现在我们通过搭建3个节点的Etcd集群，如表14-1所示，分别介绍3种方式的配置方法。

表14-1 Etcd集群环境

节点	主机名	IP
Etcd1	etcd1.example.com	192.168.3.140
Etcd2	etcd2.example.com	192.168.3.141
Etcd3	etcd3.example.com	192.168.3.142

静态服务发现是最简单的一种搭建方式，这要求事先知道Etcd集群的数目以及每个节点的地址，然后在每个Etcd节点静态地配置初始

的Etcd集群信息:

```
-initial-cluster  
etcd1=http://192.168.3.140:2380,etcd2=http://192.168.3.141:2380  
,etcd3= http://192.168.3.142:2380  
-initial-cluster-state new
```

其中, `-initial-cluster-state=new`表示这是在从无到有搭建Etcd集群。参数`-initial-cluster`描述了这个新集群中共有哪些节点, 其中每个节点用`name=url`的形式描述, 节点之间用逗号分隔, 并且指定的URL是Etcd的广播Peer地址, 即需要和`--initial-advertise-peer-urls`参数设置得一致。

分别在3个节点上启动Etcd:

- Etcd1

```
$ etcd -name etcd1 \  
-data-dir /var/lib/etcd1 \  
-listen-peer-urls http://192.168.3.140:2380 \  
-initial-advertise-peer-urls http://192.168.3.140:2380 \  
-listen-client-urls  
http://192.168.3.140:2379,http://127.0.0.1:2379 \  
-advertise-client-urls http://192.168.3.140:2379 \  
-initial-cluster  
etcd1=http://192.168.3.140:2380,etcd2=http://192.168.3.141:2380  
,etcd3=http://192.168.3.14
```

2:2380 \

-initial-cluster-state new

- Etcd2

```
$ etcd -name etcd2 \
```

```
-data-dir /var/lib/etcd2 \
```

```
-listen-peer-urls http://192.168.3.141:2380 \
```

```
-initial-advertise-peer-urls http://192.168.3.141:2380 \
```

```
-listen-client-urls
```

```
http://192.168.3.141:2379,http://127.0.0.1:2379 \
```

```
-advertise-client-urls http://192.168.3.141:2379 \
```

```
-initial-cluster
```

```
etcd1=http://192.168.3.140:2380,etcd2=http://192.168.3.141:2380
```

```
,etcd3= http://192.168.3.142:2380 \
```

```
-initial-cluster-state new
```

- Etcd3

```
$ etcd -name etcd3 \
```

```
-data-dir /var/lib/etcd3 \
```

```
-listen-peer-urls http://192.168.3.142:2380 \
```

```
-initial-advertise-peer-urls http://192.168.3.142:2380 \
```

```
-listen-client-urls
```

```
http://192.168.3.142:2379,http://127.0.0.1:2379 \
```

```
-advertise-client-urls http://192.168.3.142:2379 \
```

```
-initial-cluster
```

```
etcd1=http://192.168.3.140:2380,etcd2=http://192.168.3.141:2380
```



```
,etcd3= http://192.168.3.142:2380 \  
-initial-cluster-state new
```

动态发现适用于无法事先知道Etcd集群规模和每个节点地址的场景，包括Etcd动态发现和DNS动态发现。

Etcd动态发现

Etcd动态发现方式是利用一个已有的Etcd服务来提供服务发现，从而搭建出一个新的Etcd集群。

CoreOS官方提供了免费的Etcd发现服务（discovery.etcd.io），可以用来帮助初始化新Etcd集群。通过浏览器或者命令行curl访问地址<https://discovery.etcd.io>，可以得到一个新的集群标识URL，比如创建规模数目为3的集群标识URL：

```
$ curl -w "\n" 'https://discovery.etcd.io/new?size=3'  
https://discovery.etcd.io/e1839e0d76aa687cb7c9bafcdf420f66
```

输出的URL是搭建Etcd集群需要使用到的，可通过环境变量设置：

```
$ export ETCD_CLUSTER_DISCOVERY_URL=https://discovery.etcd.io/  
e1839e0d76aa687cb7c9bafcdf420f66
```

如果无法使用 discovery.etcd.io，也可以利用已有的 Etcd 服务，比如 Etcd服务访问地址是<http://myetcd.local>，我们同样需要创建新的集群标识 URL，首先生成一个UUID：

```
$ export ECTD_CLUSTER_UUID=$(cat /dev/urandom | base64 | tr -d
"=+/" | dd bs=40 count=1 2> /dev/null)
$ echo $ECTD_CLUSTER_UUID
Batno20LUGhFICsmuZiQgM7tGlZ84h2jLnmkLyf2
```

然后创建集群标识Key，其中设置集群规模数目为3：

```
$ curl -X PUT
http://myetcd.local/v2/keys/discovery/${ECTD_CLUSTER_UUID}/_con
fig/size -d value=3
{"action":"set","node":
{"key":"/discovery/Batno20LUGhFICsmuZiQgM7tGlZ84h2jLnmkLyf2/_co
nfig/size","value":"3","modifiedIndex":3998,"createdIndex":3998
}}
```

通过环境变量设置集群标识URL：

```
$ export
ETCD_CLUSTER_DISCOVERY_URL=http://myetcd.local/v2/keys/discover
y/${ECTD_CLUSTER_UUID}
```

在搭建新集群的时候，需要通过-discovery参数指定我们创建的URL，分别在3个节点上启动Etcd：

- Etcd1

```
$ etcd -name etcd1 \
-data-dir /var/lib/etcd1 \
-listen-peer-urls http://192.168.3.140:2380 \
```

```
-initial-advertise-peer-urls http://192.168.3.140:2380 \  
-listen-client-urls  
http://192.168.3.140:2379,http://127.0.0.1:2379  \  -advertise-  
client-urls http://192.168.3.140:2379 \  
-discovery ${ETCD_CLUSTER_DISCOVERY_URL} \  
-initial-cluster-state new
```

- Etcd2

```
$ etcd -name etcd2 \  
-data-dir /var/lib/etcd2 \  
-listen-peer-urls http://192.168.3.141:2380 \  
-initial-advertise-peer-urls http://192.168.3.141:2380 \  
-listen-client-urls  
http://192.168.3.141:2379,http://127.0.0.1:2379 \  
-advertise-client-urls http://192.168.3.141:2379 \  
-discovery ${ETCD_CLUSTER_DISCOVERY_URL} \  
-initial-cluster-state new
```

- Etcd3

```
$ etcd -name etcd3 \  
-data-dir /var/lib/etcd3 \  
-listen-peer-urls http://192.168.3.142:2380 \  
-initial-advertise-peer-urls http://192.168.3.142:2380 \  
-listen-client-urls  
http://192.168.3.142:2379,http://127.0.0.1:2379 \  
-advertise-client-urls http://192.168.3.142:2379 \  

```

```
-discovery ${ETCD_CLUSTER_DISCOVERY_URL} \  
-initial-cluster-state new
```

DNS动态发现

在DNS中，一个域名能够关联若干种资源记录，除了我们比较熟悉的A记录（用于地址查询）或者是MX记录（用于邮件服务查询），DNS还定义了SRV记录，可以用于服务发现。

Etcd支持利用DNS SRV记录实现互相发现，通过**-discovery-srv**参数设置DNS SRV域名，比如设置成example.com，然后Etcd会依次进行查询。

```
_etcd-server-ssl._tcp.example.com  
_etcd-server._tcp.example.com
```

如果_etcd-server-ssl._tcp.example.com解析成功，那么Etcd就会使用HTTPS/SSL启动。

对于我们要搭建的集群，首先要创建DNS SRV 记录：

```
$ dig +noall +answer SRV _etcd-server._tcp.example.com  
_etcd-server._tcp.example.com.  300  IN      SRV     0  0  2380  
etcd1.example.com.  
_etcd-server._tcp.example.com.  300  IN      SRV     0  0  2380  
etcd2.example.com.  
_etcd-server._tcp.example.com.  300  IN      SRV     0  0  2380  
etcd3.example.com.
```

```
$ dig +noall +answer etcd1.example.com etcd2.example.com  
etcd3.example.com  
etcd1.example.com. 300 IN A 192.168.3.140  
etcd2.example.com. 300 IN A 192.168.3.141  
etcd3.example.com. 300 IN A 192.168.3.142
```

然后分别在3个节点上启动Etcd。

- Etcd1

```
$ etcd -name etcd1 \  
-data-dir /var/lib/etcd1 \  
-listen-client-urls http://etcd1.example.com:2379 \  
-advertise-client-urls http://etcd1.example.com:2379 \  
-listen-peer-urls http://etcd1.example.com:2380 \  
-initial-advertise-peer-urls http://etcd1.example.com:2380 \  
-discovery-srv example.com \  
-initial-cluster-state new
```

- Etcd2

```
$ etcd -name etcd2 \  
-data-dir /var/lib/etcd2 \  
-listen-client-urls http://etcd2.example.com:2379 \  
-advertise-client-urls http://etcd2.example.com:2379 \  
-listen-peer-urls http://etcd2.example.com:2380 \  
-initial-advertise-peer-urls http://etcd2.example.com:2380 \  

```

```
-discovery-srv example.com \  
-initial-cluster-state new
```

- Etcd3

```
$ etcd -name etcd3 \  
-data-dir /var/lib/etcd3 \  
-listen-client-urls http://etcd3.example.com:2379 \  
-advertise-client-urls http://etcd3.example.com:2379 \  
-listen-peer-urls http://etcd3.example.com:2380 \  
-initial-advertise-peer-urls http://etcd3.example.com:2380 \  
-discovery-srv example.com \  
-initial-cluster-state new
```

14.3.2.2 集群管理

Etcd集群搭建完成后，可以查询集群成员：

```
$ etcdctl member list  
f042ea167d8f2828: name=etcd1 peerURLs=http://192.168.3.140:2380  
clientURLs=http://192.168.3.140:2379  
156ce626171618a6: name=etcd2 peerURLs=http://192.168.3.141:2380  
clientURLs=http://192.168.3.141:2379  
c99b1511c3ade2bb: name=etcd3 peerURLs=http://192.168.3.142:2380  
clientURLs=http://192.168.3.142:2379
```

以及集群的健康状态：

```
$ etcdctl cluster-health
```

```
member f042ea167d8f2828 is healthy: got healthy result from  
http://192.168.3.140:2379
```

```
member 156ce626171618a6 is healthy: got healthy result from  
http://192.168.3.141:2379
```

```
member c99b1511c3ade2bb is healthy: got healthy result from  
http://192.168.3.142:2379
```

```
cluster is healthy
```

增加成员

添加成员信息:

```
$ etcdctl member add etcd4 http://192.168.3.143:2380
```

```
added member 4d906f7a642c3f23 to cluster
```

```
ETCD_NAME="etcd4"
```

```
ETCD_INITIAL_CLUSTER="etcd1=http://192.168.3.140:2380      ,
```

```
etcd2=http://192.168.3.141:2380                             ,
```

```
etcd3=http://192.168.3.142:2380                             ,
```

```
etcd4=http://192.168.3.143:2380"
```

```
ETCD_INITIAL_CLUSTER_STATE=existing
```

添加成功后返回的信息中提示设置环境变量 `etcd_name` 、`etcd_initial_cluster` 和 `etcd_initial_cluster_state` , 这将在启动新的 Etcd 成员时使用:

```
$ etcd -name etcd4 \  
-data-dir /var/lib/etcd4 \  
-listen-client-urls http://192.168.3.143:2379 \  
-advertise-client-urls http://192.168.3.143:2379 \  
-listen-peer-urls http://192.168.3.143:2380 \  
-initial-advertise-peer-urls http://192.168.3.143:2380 \  
-initial-cluster $ETCD_INITIAL_CLUSTER \  
-initial-cluster-state existing
```

最后，可以查询到新的成员列表：

```
$ etcdctl member list  
f042ea167d8f2828: name=etcd1 peerURLs=http://192.168.3.140:2380  
clientURLs=http://192.168.3.140:2379  
156ce626171618a6: name=etcd2 peerURLs=http://192.168.3.141:2380  
clientURLs=http://192.168.3.141:2379  
c99b1511c3ade2bb: name=etcd3 peerURLs=http://192.168.3.142:2380  
clientURLs=http://192.168.3.142:2379  
4d906f7a642c3f23: name=etcd4 peerURLs=http://192.168.3.143:2380  
clientURLs=http://192.168.3.143:2379
```

删除成员

```
$ etcdctl member remove 4d906f7a642c3f23  
Removed member 4d906f7a642c3f23 from cluster
```

```
$ etcdctl member list  
f042ea167d8f2828: name=etcd1 peerURLs=http://192.168.3.140:2380
```



```
clientURLs=http://192.168.3.140:2379
156ce626171618a6: name=etcd2 peerURLs=http://192.168.3.141:2380
clientURLs=http://192.168.3.141:2379
c99b1511c3ade2bb: name=etcd3 peerURLs=http://192.168.3.142:2380
clientURLs=http://192.168.3.142:2379
```

更新成员

```
$ etcdctl member update f042ea167d8f2828
http://etcd1.example.com:2380
Updated member with ID f042ea167d8f2828 in cluster
```

```
$ etcdctl member list
f042ea167d8f2828: name=etcd1
peerURLs=http://etcd1.example.com:2380
clientURLs=http://192.168.3.140:2379
156ce626171618a6: name=etcd2 peerURLs=http://192.168.3.141:2380
clientURLs=http://192.168.3.141:2379
c99b1511c3ade2bb: name=etcd3 peerURLs=http://192.168.3.142:2380
clientURLs=http://192.168.3.142:2379
```

迁移成员

当需要迁移成员的时候，实际上等效于删除一个旧成员，再新增一个新成员，让数据自动切换，但是如果数据过大（大于50MB），会影响集群的状态。更安全的做法是人为地迁移数据，比如现在需要将 Etcd 成员 etcd3 从 机器 1 （ 192.168.3.142 ） 迁移 机器 2 （192.168.3.143），步骤如下。

- 首先在机器1上停止Etcd进程，打包数据并复制到机器2:

```
$ kill `pgrep etcd`  
$ tar -cvzf etcd3.data.tar.gz /var/lib/etcd3  
$ scp etcd3.data.tar.gz 192.168.3.143:~/
```

紧接着更新Etcd成员:

```
$ etcdctl member update c99b1511c3ade2bb  
http://192.168.3.143:2380  
Updated member with ID c99b1511c3ade2bb in cluster
```

然后在机器2上启动Etcd:

```
$ tar -xvzf etcd3.data.tar.gz -C /var/lib/etcd3  
$ etcd -name etcd3 \  
-data-dir /var/lib/etcd3 \  
-listen-peer-urls http://192.168.3.143:2380 \  
-initial-advertise-peer-urls http://192.168.3.143:2380 \  
-listen-client-urls  
http://192.168.3.143:2379,http://127.0.0.1:2379 \  
-advertise-client-urls http://192.168.3.143:2379
```

14.3.3 Etcd Proxy模式

Etcd可以设置为Proxy模式，此时Etcd Proxy并不是直接加入到数据强一致性的Etcd集群中，所以Etcd Proxy并没有增加集群的可靠性，

当然也没有降低集群的写入性能。Etcd Proxy只是作为一个反向代理把客户的请求转发给可用的Etcd集群。一个典型的应用场景是，在客户端机器本地运行Etcd Proxy来对接真正的Etcd服务端集群，对客户端应用来说可以直接访问本地的Etcd Proxy（<http://127.0.0.1:2379>），而不用感知Etcd服务端集群的变化，因为Etcd Proxy会维护同Etcd服务端集群的同步。

启动一个Etcd Proxy，对接Etcd集群：

- 静态发现

```
$ etcd -proxy on \  
-listen-client-urls http://127.0.0.1:4001 \  
-initial-cluster  
etcd1=http://192.168.3.140:2380,etcd2=http://192.168.3.141:2380  
, etcd3=http://192.168.3.142:2380
```

- Etcd动态发现

```
$ etcd -proxy on \  
-listen-client-urls http://127.0.0.1:4001 \  
-discovery ${DISCOVERY_URL}
```

- DNS动态发现

```
$ etcd --proxy on \  
-listen-client-urls http://127.0.0.1:4001 \  
-discovery-srv example.com
```

14.3.4 Etcd的安全模式

Etcd中的通信包括Client-to-Server和Peer-to-Peer，支持SSL/TLS加密和Client Certificate Authentication认证。

Client-to-Server通信

- 开启HTTPS

开启HTTPS，需要CA证书（server-ca.crt）和该CA签发的服务端密钥对（server.crt/ server.key）。运行Etcd:

```
$ etcd -name etcd1 \  
-cert-file=/path/to/server.crt -key-file=/path/to/server.key \  
-advertise-client-urls=https://127.0.0.1:2379 \  
-listen-client-urls=https://127.0.0.1:2379
```

Etcd Client通过HTTPS访问Etcd Server时就需提供CA证书（server-ca.crt）：

```
$ etcdctl --ca-file=/path/to/server-ca.crt -C  
https://127.0.0.1:2379 cluster-health  
member ce2a822cea30bfca is healthy: got healthy result from  
https://127.0.0.1:2379  
cluster is healthy
```

- 开启Client Certificate Authentication

在开启HTTPS后，可以开启Client Certificate Authentication。这样一来，Etcd Client在访问Etcd Server的时候就需要提供Client Certificate进行认证，认证成功才能有权限访问。

同样的，需要CA证书（client-ca.crt）和该CA签发的客户端密钥对（client.crt/client.key），签发客户端密钥对和签发服务端的CA可以一样，这实际上并不冲突，因为一个CA可以同时签发服务端密钥对和客户端密钥对。运行Etcd:

```
$ etcd -name etcd1 -data-dir etcd1 \  
-client-cert-auth -trusted-ca-file=/path/to/client-ca.crt \  
-cert-file=/path/to/server.crt -key-file=/path/to/server.key \  
-advertise-client-urls https://127.0.0.1:2379 \  
-listen-client-urls https://127.0.0.1:2379
```

如Etcd Client按照之前的命令访问Etcd Server，会报错:

```
$ etcdctl --ca-file=/path/to/server-ca.crt -C  
https://127.0.0.1:2379 cluster-health  
cluster may be unhealthy: failed to list members  
Error: client: etcd cluster is unavailable or misconfigured  
error #0: remote error: bad certificate
```

因为Etcd Client需要提供Client Certificate，才能认证成功:

```
$ etcdctl --ca-file=/path/to/server-ca.crt \  
--key-file=/path/to/client.key --cert-file=/path/to/client.crt \  
-C https://127.0.0.1:2379 \  

```

cluster-health

```
member ce2a822cea30bfca is healthy: got healthy result from  
https://127.0.0.1:2379  
cluster is healthy
```

Peer-to-Peer通信

- 开启HTTPS和Client Certificate Authentication

类似的，Etcd Peer之间的通信开启HTTPS和Client Certificate Authentication，需要准备CA证书（ca.crt），为每个Peer签发一个密钥对（server1.key和server1.crt，server2.key和server2.crt）。

- Etcd1

```
$ etcd -name etcd1 \  
-peer-client-cert-auth -peer-trusted-ca-file=/path/to/ca.crt \  
-peer-cert-file=/path/to/server1.crt -peer-key-  
file=/path/to/server1.key \  
-listen-peer-urls=https://192.168.3.140:2380 \  
-initial-advertise-peer-urls=https://192.168.3.140:2380 \  
-initial-cluster  
etcd1=https://192.168.3.140:2380,etcd2=https://192.168.3.141:23  
80 \  
-initial-cluster-state new
```

- Etcd2

```
$ etcd -name etcd2 \  
-peer-client-cert-auth -peer-trusted-ca-file=/path/to/ca.crt \  
-peer-cert-file=/path/to/server2.crt -peer-key-  
file=/path/to/server2.key \  
-listen-peer-urls=https://192.168.3.141:2380 \  
-initial-advertise-peer-urls=https://192.168.3.141:2380 \  
-initial-cluster  
etcd1=http://192.168.3.140:2380,etcd2=http://192.168.3.141:2380  
, etcd3=http://192.168.3.142:2380 \  
-initial-cluster  
etcd1=https://192.168.3.140:2380,etcd2=https://192.168.3.141:23  
80 \  
-initial-cluster-state new
```

第15章

Mesos

Mesos是一个成熟的架构，一方面提供了同Kubernetes类似的容器集群管理方案；另一方面，Mesos正积极同Kubernetes进行整合，Mesos和Kubernetes亦敌亦友。本章将介绍Mesos，以及Marathon和K8SM，最后说明如何使用这3个项目。

15.1 Mesos介绍

Apache Mesos是由加州大学伯克利分校的AMPLab首先开发的一款开源集群管理软件，支持Hadoop、Elasticsearch、Spark、Storm和Kafka等架构。其开源性越来越受到一些大型云计算公司的青睐，Twitter和Airbnb公司已经将其大规模应用在其数据中心中了。现在，一家初创公司Mesosphere将Mesos定位为数据中心操作系统（Data Center Operating System，DCOS），使之步入主流。

Mesos在多种不同类型的工作之间共享机器（或者节点）的可用资源，如图15-1所示。Mesos可以看作是数据中心的内核，提供所有节点资源的统一视图，所起的作用类似于操作系统内核在单台机器上的作用，可以无缝地访问多节点资源。

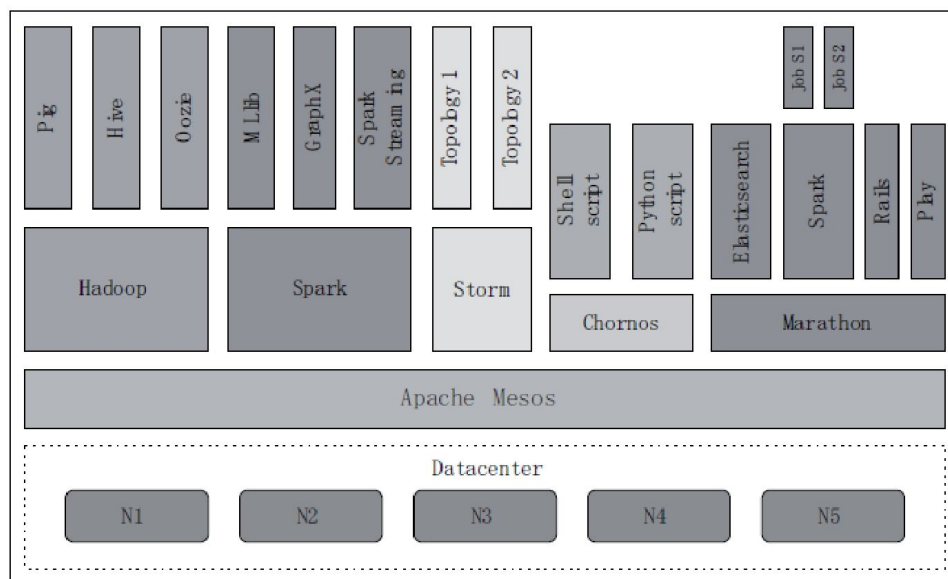


图15-1 Mesos的层次

15.2 Mesos的架构

Mesos的架构如图15-2所示，主要的角色有Mesos Master、Mesos Slave、Framework和Executor。

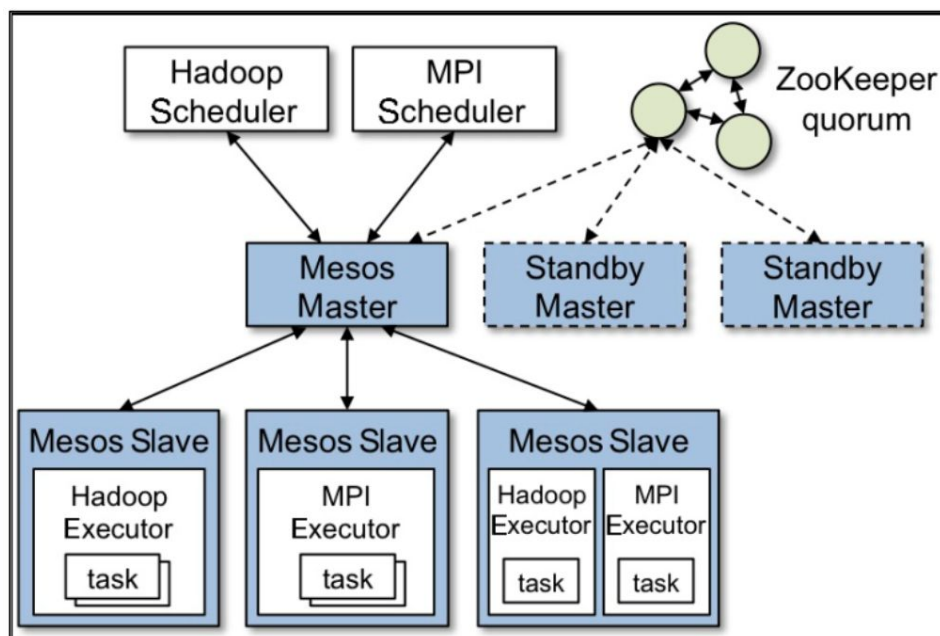


图15-2 Mesos架构

- Mesos Master是整个系统的核心，负责管理接入Mesos的各个Framework和Mesos Slave，并将Mesos Slave上的资源按照某种策略分配给Framework。

- Mesos Slave负责接收并执行来自Mesos Master的命令、管理节点上的任务，并为各个任务分配资源。Mesos Slave将自己的资源量发送给Mesos Master，由Mesos Master决定将资源分配给哪个Framework，并且当任务运行时，Mesos Slave会将任务放到包含固定资源的Linux容器中运行，以达到资源隔离的效果。

- Framework是指外部的计算框架，如Hadoop、Elasticsearch、Spark、Storm和Kafka等，这些计算框架可通过注册的方式接入Mesos，以便Mesos进行统一管理和资源分配。Mesos要求可接入的Framework必须有一个调度器模块，该调度器负责Framework内部的任务调度。当一个Framework想要接入Mesos时，需要修改自己的调度

器，以便向Mesos注册，并获取Mesos分配给自己的资源，这样再由调度器将这些资源分配给Framework中的任务。也就是说，整个Mesos系统采用了双层调度框架：第一层，由Mesos将资源分配给Framework；第二层，Framework的调度器将资源分配给自己内部的任务。

- **Executor** 主要用于启动Framework内部的任务。由于不同的Framework启动任务的接口或者方式不同，当一个新的Framework要接入Mesos时，需要编写一个Executor，告诉Mesos如何启动该Framework中的任务。

15.3 Marathon和K8SM介绍

15.3.1 Marathon

Marathon是一个Mesos Framework，正如其名字“马拉松”，用来支持运行长期服务，比如Web应用等。Marathon能够保证这些服务的高可用性，也就是说，当某台机器发生故障时，能够在其他机器上启动服务。Marathon是集群的分布式Init.d，能够原样运行任何Linux二进制发布版本，如Tomcat、Play等，它也是一种私有的PaaS，提供REST API服务，实现服务的发现，有授权和SSL、配置约束，通过HAProxy实现服务发现和负载均衡。更重要的是，Marathon已经集成了Docker，Mesos+Marathon是比较成熟的容器集群管理解决方案，已经在众多知名企业的生产环境中使用。图15-3所示为Marathon的结构图。

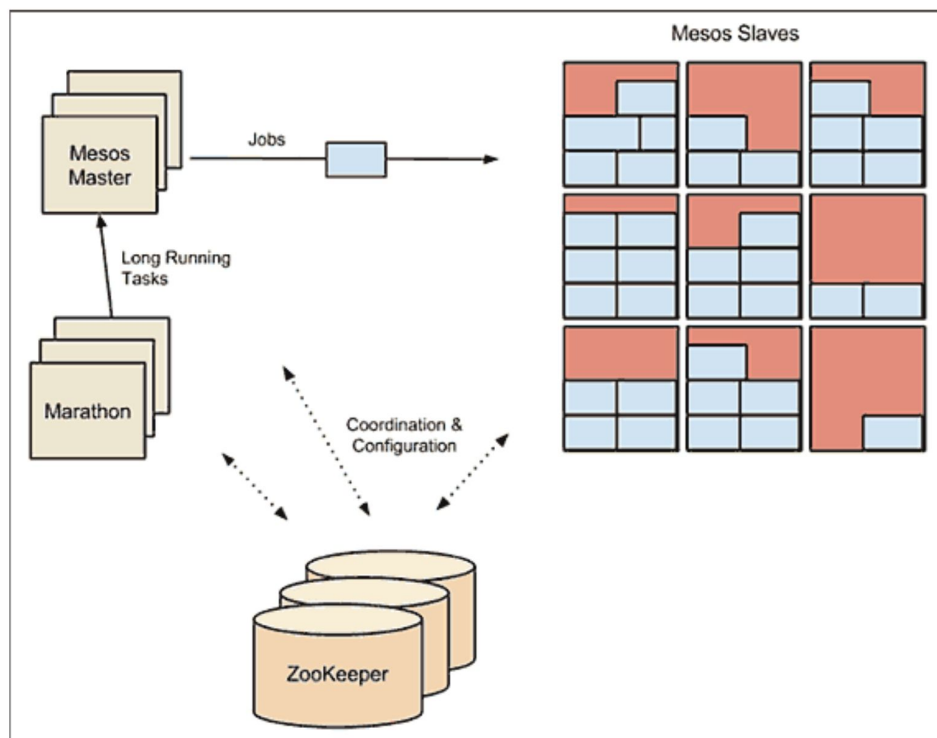


图15-3 Marathon结构

15.3.2 K8SM

K8SM（Kubernetes-Mesos）是一个将Kubernetes整合到Mesos中的项目，作为Mesos Framework，能够将Kubernetes处理并运行在Mesos之上，且可以同任意数量的其他Mesos框架（包括Marathon、Spark、Kafka以及Jenkins等）实现同地协作，从而共享来自同一套集群中的各类资源。

Kubernetes是一个轻量级的容器管理平台，可深度和灵活地进行容量编排，而Mesos是分布式系统内核，它可以将不同的机器整合在一个逻辑计算机中，有着非常优秀的资源调度策略，可以认为Mesos

和Kubernetes的愿景差不多，而K8SM则整合了两者，K8SM的架构如图15-4所示。

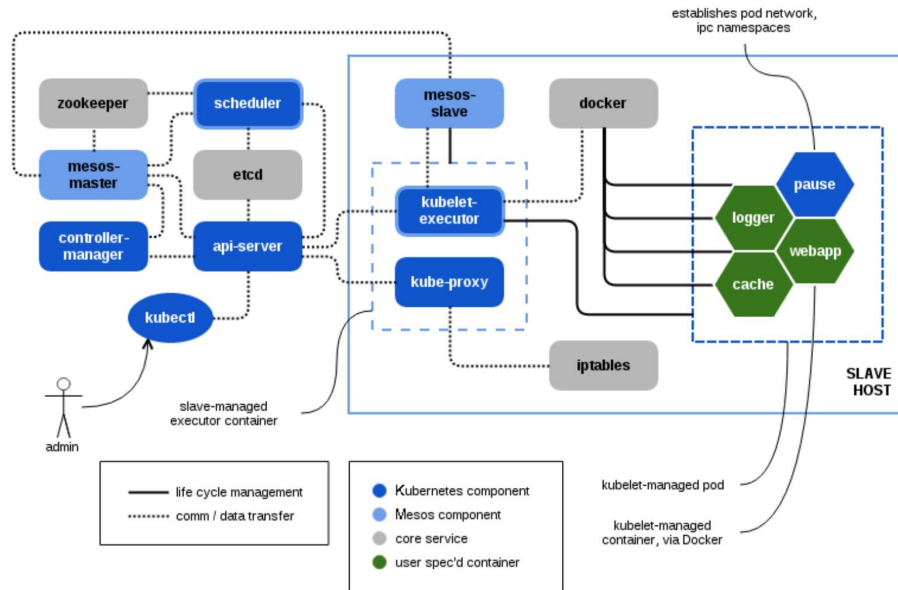


图15-4 K8SM架构

15.4 Mesos实践

15.4.1 运行Mesos

1. 运行Zookeeper。

```
$ docker run -d \  
--name zookeeper --net host \  
mesoscloud/zookeeper:3.4.6
```

2. 运行Mesos Master。

```
$ docker run -d \  
-e MESOS_HOSTNAME=mesos-master \  
-e MESOS_IP=<mesos-master-ip> \  
-e MESOS_QUORUM=1 \  
-e MESOS_ZK=zk://<zookeeper-ip>:2181/mesos \  
--name mesos-master --net host \  
mesoscloud/mesos-master:0.24.1
```

3. 运行Mesos Slave。

```
$ docker run -d \  
-e MESOS_HOSTNAME=mesos-slave \  
-e MESOS_IP=<mesos-slave-ip> \  
-e MESOS_MASTER=zk://<zookeeper-ip>:2181/mesos \  
-v /sys/fs/cgroup:/sys/fs/cgroup \  
-v /var/run/docker.sock:/var/run/docker.sock \  
--name mesos-slave --net host --privileged \  
mesoscloud/mesos-slave:0.24.1
```

提示

Mesos Slave节点上需要预先安装好Docker(V1.9.1)。

4. 待Mesos运行正常后，可以通过默认5050端口访问Mesos Web界面，如图15-5所示。

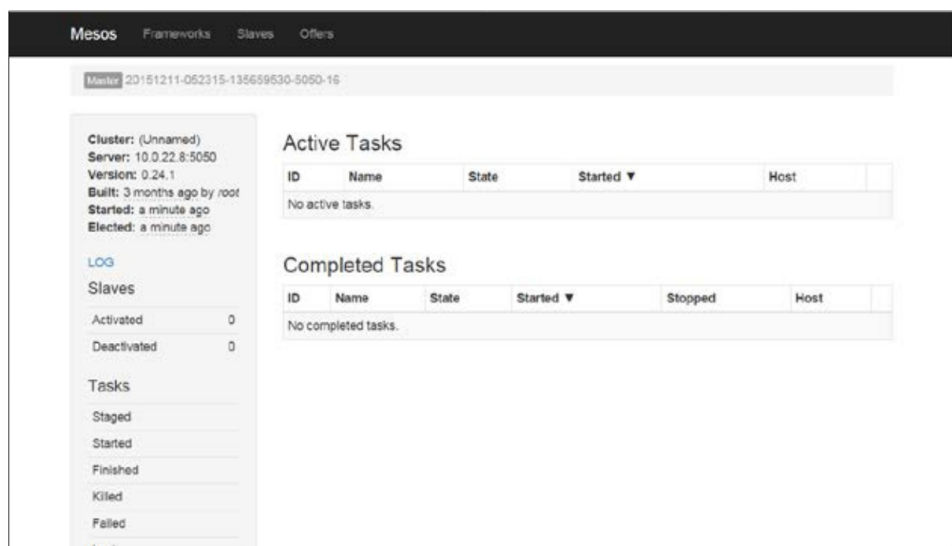


图15-5 Mesos Web界面

同时可以查询到Mesos Slave，如图15-6所示。

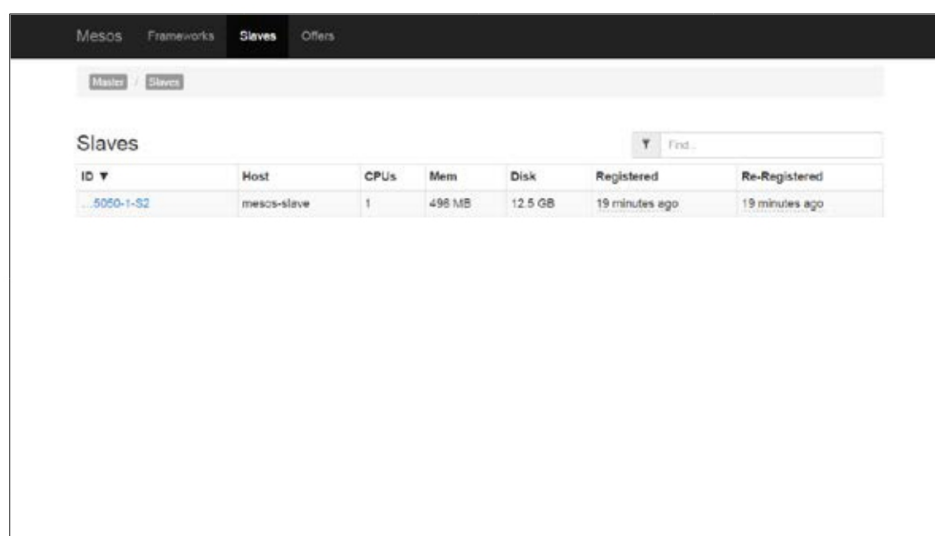


图15-6 查询Mesos Slave

15.4.2 运行Marathon

1. 运行Marathon。

```
$ docker run -d \  
-e MARATHON_HOSTNAME=marathon \  
-e MARATHON_HTTP_ADDRESS=<marathon-ip> \  
-e MARATHON_MASTER=zk://<zookeeper-ip>:2181/mesos \  
-e MARATHON_ZK=zk://<zookeeper-ip>:2181/marathon \  
--name marathon --net host \  
mesoscloud/marathon:0.11.0
```

2. Marathon运行成功后，在Mesos界面上可以查询到Marathon注册信息，如图15-7所示。同时可以通过8080端口访问Marathon Web界面，如图15-8所示。

3. 通过调用Marathon REST API创建应用。

```
$ curl -v \  
-X POST -H "Content-Type: application/json" \  
--data "@app.json" \  
http://<marathon-ip>:8080/v2/apps
```

其中app.json是应用的定义文件，内容如下：

```
{  
  "id": "web-app",  
  "cmd": "python3 -m http.server 8080",  
  "cpus": 0.5,  
  "mem": 32.0,  
  "instances": 1,  
  "container": {
```



```

"type": "DOCKER",
"docker": {
  "image": "python:3",
  "network": "BRIDGE",
  "portMappings": [
    { "containerPort": 8080, "hostPort": 0 }
  ]
}
}
}

```

Mesos

Frameworks

Slaves

Offers

Mesos

Frameworks

Active Frameworks

▼

Find...

ID ▼	Host	User	Name	Active Tasks	CPUs	Mem	Max Share	Registered	Re-Registered
5050-1-0000	marathon	root	marathon	0	0	0 B	0%	19 minutes ago	-

Terminated Frameworks

ID ▼	Host	User	Name	Registered	Unregistered
------	------	------	------	------------	--------------

图15-7 查询Marathon注册信息

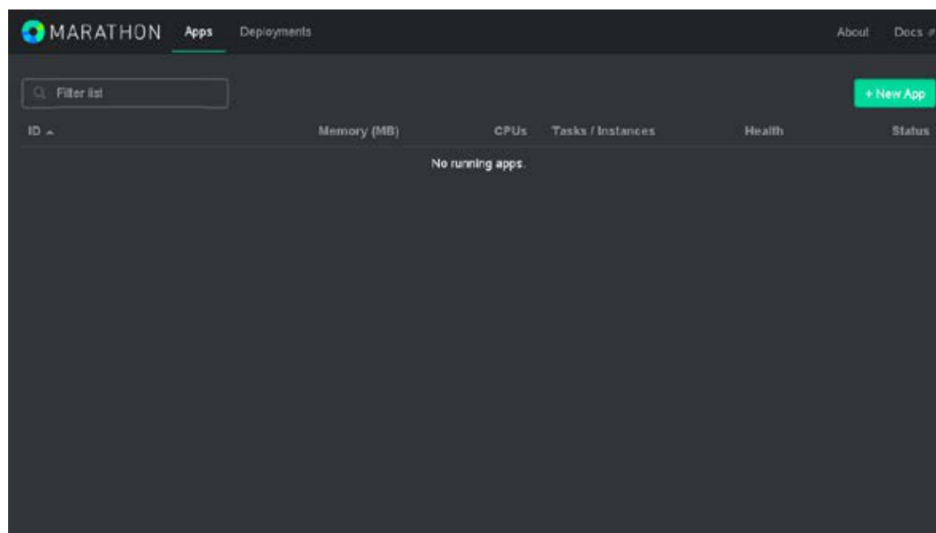


图15-8 Marathon Web界面

在应用定义文件中首先指定了应用的ID、资源规格和运行实例数，在container属性中配置了Docker容器的运行参数。

应用创建成功后，Mesos Slave将会运行起配置的Docker容器。另外，在Marathon Web界面可以查询到应用详情，如图15-9所示。

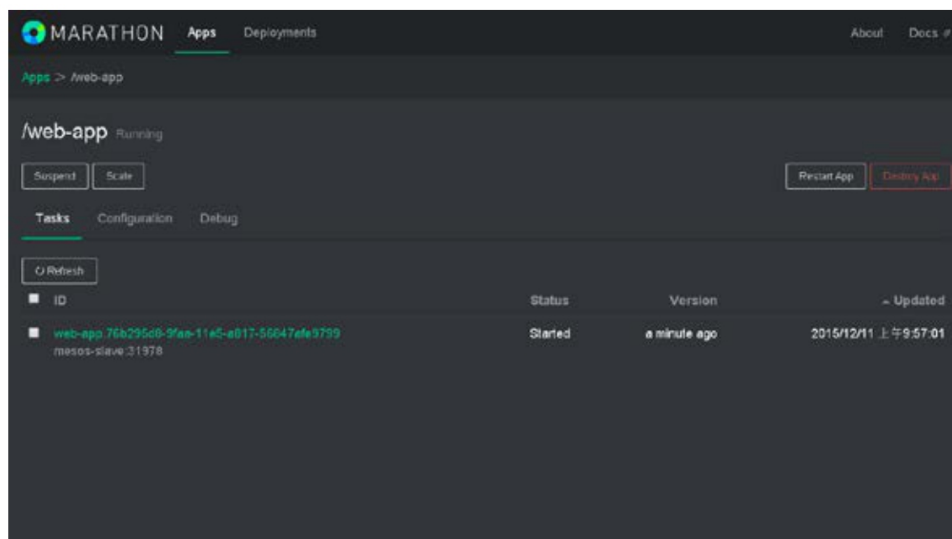


图15-9 查询应用详情

15.4.3 运行K8SM

1. 构建K8SM。

```
$ git clone https://github.com/kubernetes/kubernetes
$ export KUBERNETES_CONTRIB=mesos
$ make
```

编译成功后，可执行程序在_output/local/go/bin目录下可以设置到系统环境变量中：

```
$ export PATH="$(pwd)/_output/local/go/bin:$PATH"
```

2. 启动K8SM。

设置Mesos Master的地址：

```
$ export MESOS_MASTER=<mesos-master-ip>:5050
```

然后配置文件mesos-cloud.conf：

```
$ cat <<EOF >mesos-cloud.conf
[mesos-cloud]
    mesos-master          = ${MESOS_MASTER}
EOF
```

设置环境变量：

```
$ export KUBERNETES_MASTER_IP=$(hostname -i)
$ export KUBERNETES_MASTER=http://${KUBERNETES_MASTER_IP}:8888
```

首先运行Etcd:

```
$ docker run -d \  
-p 4001:4001 -p 7001:7001 \  
--hostname $(uname -n) --name etcd \  
quay.io/coreos/etcd:v2.0.12 \  
--listen-client-urls http://0.0.0.0:4001 \  
--advertise-client-urls http://0.0.0.0:4001
```

然后运行K8SM组件:

```
$ km apiserver \  
--address=${KUBERNETES_MASTER_IP} \  
--etcd-servers=http://127.0.0.1:4001 \  
--service-cluster-ip-range=10.10.10.0/24 \  
--port=8888 \  
--cloud-provider=mesos \  
--cloud-config=mesos-cloud.conf \  
--secure-port=0 \  
--v=1 >apiserver.log 2>&1 &  
  
$ km controller-manager \  
--master=http://${KUBERNETES_MASTER_IP}:8888 \  
--cloud-provider=mesos \  
--cloud-config=./mesos-cloud.conf \  
--v=1 >controller.log 2>&1 &  
  
$ km scheduler \  

```

```

--address=${KUBERNETES_MASTER_IP} \
--mesos-master=${MESOS_MASTER} \
--etcd-servers=http://127.0.0.1:4001 \
--mesos-user=root \
--api-servers=${KUBERNETES_MASTER_IP}:8888 \
--cluster-dns=10.10.10.10 \
--cluster-domain=cluster.local \
--v=2 >scheduler.log 2>&1 &

```

3. K8SM运行成功后可以进行查询。

```

$ kubectl -s http://${KUBERNETES_MASTER_IP}:8888 get services

```

NAME	CLUSTER_IP	EXTERNAL_IP	PORT(S)
SELECTOR	AGE		
k8sm-scheduler	10.10.10.132	<none>	10251/TCP
<none>	9s		
kubernetes	10.10.10.1	<none>	443/TCP
<none>	40s		

同时在Mesos Web界面上可以查询到K8SM注册信息，如图15-10所示。

Mesos

Frameworks

Slaves

Offers

Mesos / Frameworks

Active Frameworks

▼

Find...

ID ▼	Host	User	Name	Active Tasks	CPUs	Mem	Max Share	Registered	Re-Registered
...5050-1-0000	k8sm-master	root	Kubernetes	0	1	496 MB	100%	2 minutes ago	-
...5050-1-0000	marathon	root	marathon	0	0	0 B	0%	2 hours ago	-

Terminated Frameworks

ID ▼	Host	User	Name	Registered	Unregistered
------	------	------	------	------------	--------------

图15-10 查询K8SM注册信息

4. 使用K8SM运行Pod。

```
$ kubectl -s http://${KUBERNETES_MASTER_IP}:8888 run nginx --
image=nginx
replicationcontroller "nginx" created
```

当有Pod创建后，K8SM Scheduler向Mesos申请资源，然后K8SM Scheduler根据Mesos提供的资源绑定Pod到指定的Mesos Slave。这时候，K8SM自动会在Mesos Slave上运行K8SM Executor，其中包括Kubelet和Kube Proxy组件。相应的，该Mesos Slave也会被作为Kubernetes Node：

```
$ kubectl -s http://${KUBERNETES_MASTER_IP}:8888 get node

NAME                                                                 LABELS
STATUS      AGE
mesos-slave   kubernetes.io/hostname=mesos-slave   Ready
22s
```

最后在Mesos Slave上，K8SM将会运行起Pod:

```
$ kubectl -s http://${KUBERNETES_MASTER_IP}:8888 get pod --  
selector run=nginx
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-1dk7d	1/1	Runing	0	1m

Broadview
www.broadview.com.cn

Kubernetes

权威指南

第2版

龚正 吴治辉 王伟 / 主编
崔秀龙 阎健勇 / 参编

从Docker到Kubernetes
实践全接触



下一代分布式架构的王者

Kubernetes: The Definitive Guide



中国工信出版集团



电子工业出版社
Publishing House of Electronics Industry
地址: 北京市西城区德胜大街甲6号
邮编: 100085
电话: (010) 88379636
网址: http://www.phei.com.cn

版权信息

书名：Kubernetes权威指南：从Docker到Kubernetes实践全接触（第2版）

作者：龚正等编著

出版社：电子工业出版社

ISBN：978-7-121-29941-4

定价：99.00

版权所有·侵权必究

目 录

[第1版推荐序](#)

[自序](#)

[第1章 Kubernetes入门](#)

[1.1 Kubernetes是什么](#)

[1.2 为什么要用Kubernetes](#)

[1.3 从一个简单的例子开始](#)

[1.3.1 环境准备](#)

[1.3.2 启动MySQL服务](#)

[1.3.3 启动Tomcat应用](#)

[1.3.4 通过浏览器访问网页](#)

[1.4 Kubernetes基本概念和术语](#)

[1.4.1 Master](#)

[1.4.2 Node](#)

[1.4.3 Pod](#)

[1.4.4 Label（标签）](#)

[1.4.5 Replication Controller（RC）](#)

[1.4.6 Deployment](#)

[1.4.7 Horizontal Pod Autoscaler（HPA）](#)

[1.4.8 Service（服务）](#)

[1.4.9 Volume（存储卷）](#)

[1.4.10 Persistent Volume](#)

[1.4.11 Namespace（命名空间）](#)

[1.4.12 Annotation（注解）](#)

[1.4.13 小结](#)

[第2章 Kubernetes实践指南](#)

[2.1 Kubernetes安装与配置](#)

[2.1.1 安装Kubernetes](#)

[2.1.2 配置和启动Kubernetes服务](#)

[2.1.3 Kubernetes集群的安全设置](#)

[2.1.4 Kubernetes的版本升级](#)

[2.1.5 内网中的Kubernetes相关配置](#)

[2.1.6 Kubernetes核心服务配置详解](#)

[2.1.7 Kubernetes集群网络配置方案](#)

[2.2 kubectl命令行工具用法详解](#)

[2.2.1 kubectl用法概述](#)

[2.2.2 kubectl子命令详解](#)

[2.2.3 kubectl参数列表](#)

[2.2.4 kubectl输出格式](#)

[2.2.5 kubectl操作示例](#)

[2.3 Guestbook示例： Hello World](#)

[2.3.1 创建redis-master RC和服务](#)

[2.3.2 创建redis-slave RC和服务](#)

[2.3.3 创建frontend RC和服务](#)

[2.3.4 通过浏览器访问frontend页面](#)

[2.4 深入掌握Pod](#)

[2.4.1 Pod定义详解](#)

[2.4.2 Pod的基本用法](#)

[2.4.3 静态Pod](#)

[2.4.4 Pod容器共享Volume](#)

[2.4.5 Pod的配置管理](#)

[2.4.6 Pod生命周期和重启策略](#)

[2.4.7 Pod健康检查](#)

[2.4.8 玩转Pod调度](#)

[2.4.9 Pod的扩容和缩容](#)

[2.4.10 Pod的滚动升级](#)

[2.5 深入掌握Service](#)

[2.5.1 Service定义详解](#)

[2.5.2 Service基本用法](#)

[2.5.3 集群外部访问Pod或Service](#)

[2.5.4 DNS服务搭建指南](#)

[2.5.5 Ingress: HTTP 7层路由机制](#)

[第3章 Kubernetes核心原理](#)

[3.1 Kubernetes API Server原理分析](#)

[3.1.1 Kubernetes API Server概述](#)

[3.1.2 独特的Kubernetes Proxy API接口](#)

[3.1.3 集群功能模块之间的通信](#)

[3.2 Controller Manager原理分析](#)

[3.2.1 ReplicationController](#)

[3.2.2 NodeController](#)

[3.2.3 ResourceQuotaController](#)

[3.2.4 NamespaceController](#)

[3.2.5 Service Controller与Endpoint Controller](#)

[3.3 Scheduler原理分析](#)

[3.4 kubelet运行机制分析](#)

[3.4.1 节点管理](#)

[3.4.2 Pod管理](#)

[3.4.3 容器健康检查](#)

[3.4.4 cAdvisor资源监控](#)

[3.5 kube-proxy运行机制分析](#)

[3.6 深入分析集群安全机制](#)

[3.6.1 API Server认证](#)

[3.6.2 API Server授权](#)

[3.6.3 Admission Control准入控制](#)

[3.6.4 Service Account](#)

[3.6.5 Secret私密凭据](#)

[3.7 网络原理](#)

[3.7.1 Kubernetes网络模型](#)

[3.7.2 Docker的网络基础](#)

[3.7.3 Docker的网络实现](#)

[3.7.4 Kubernetes的网络实现](#)

[3.7.5 开源的网络组件](#)

[3.7.6 网络实战](#)

[第4章 Kubernetes开发指南](#)

[4.1 REST简述](#)

[4.2 Kubernetes API详解](#)

[4.2.1 Kubernetes API概述](#)

[4.2.2 API版本](#)

[4.2.3 API详细说明](#)

[4.2.4 API响应说明](#)

[4.3 使用Java程序访问Kubernetes API](#)

[4.3.1 Jersey](#)

[4.3.2 Fabric8](#)

[4.3.3 使用说明](#)

[第5章 Kubernetes运维指南](#)

[5.1 Kubernetes集群管理指南](#)

[5.1.1 Node的管理](#)

[5.1.2 更新资源对象的Label](#)

[5.1.3 Namespace: 集群环境共享与隔离](#)

[5.1.4 Kubernetes资源管理](#)

[5.1.5 Kubernetes集群高可用部署方案](#)

[5.1.6 Kubernetes集群监控](#)

[5.1.7 kubelet的垃圾回收（GC）机制](#)

[5.2 Kubernetes高级案例](#)

[5.2.1 ElasticSearch日志搜集查询和展现案例](#)

[5.2.2 Cassandra集群部署案例](#)

[5.3 Trouble Shooting指导](#)

[5.3.1 查看系统Event事件](#)

[5.3.2 查看容器日志](#)

[5.3.3 查看Kubernetes服务日志](#)

[5.3.4 常见问题](#)

[5.3.5 寻求帮助](#)

[5.4 Kubernetes v1.3开发中的新功能](#)

[5.4.1 PetSet（有状态的容器）](#)

[5.4.2 Init Container（初始化容器）](#)

[5.4.3 ClusterFederation（集群联邦）](#)

[第6章 Kubernetes源码导读](#)

[6.1 Kubernetes源码结构和编译步骤](#)

[6.2 kube-apiserver进程源码分析](#)

[6.2.1 进程启动过程](#)

[6.2.2 关键代码分析](#)

[6.2.3 设计总结](#)

[6.3 kube-controller-manager进程源码分析](#)

[6.3.1 进程启动过程](#)

[6.3.2 关键代码分析](#)

[6.3.3 设计总结](#)

[6.4 kube-scheduler进程源码分析](#)

[6.4.1 进程启动过程](#)

[6.4.2 关键代码分析](#)

[6.4.3 设计总结](#)

[6.5 kubelet进程源码分析](#)

[6.5.1 进程启动过程](#)

[6.5.2 关键代码分析](#)

[6.5.3 设计总结](#)

[6.6 kube-proxy进程源码分析](#)

[6.6.1 进程启动过程](#)

[6.6.2 关键代码分析](#)

[6.6.3 设计总结](#)

[6.7 kubectl进程源码分析](#)

[6.7.1 kubectl create命令](#)

[6.7.2 rolling-update命令](#)

[后记](#)

[返回总目录](#)

第1版推荐序

经过作者们多年的实践经验积累及近一年的精心准备，本书终于与我们大家见面了。我有幸作为首批读者，提前见证和学习了在云时代引领业界技术方向的Kubernetes和Docker的最新动态。

从内容上讲，本书从一个开发者的角度去理解、分析和解决问题：从基础入门到架构原理，从运行机制到开发源码，再从系统运维到应用实践，讲解全面。本书图文并茂，内容丰富，由浅入深，对基本原理阐述清晰，对程序源码分析透彻，对实践经验体会深刻。

我认为本书值得推荐的原因有以下几点。

首先，作者的所有观点和经验，均是在多年建设、维护大型应用系统的过程中积累形成的。例如，读者通过学习书中的Kubernetes运维指南和高级应用实践案例章节的内容，不仅可以直接提高开发技能，还可以解决在实践过程中经常遇到的各种关键问题。书中的这些内容具有很高的借鉴和推广意义。

其次，通过大量的实例操作和详尽的源码解析，本书可以帮助读者进一步深刻理解Kubernetes的各种概念。例如书中“Java访问Kubernetes API”的几种方法，读者参照其中的案例，只要稍做修改，再结合实际的应用需求，就可以用于正在开发的项目中，达到事半功

倍的效果，有利于有一定Java基础的专业人士快速学习Kubernetes的各种细节和实践操作。

再次，为了让初学者快速入门，本书配备了即时在线交流工具和专业后台技术支持团队。如果你在开发和应用的过程中遇到各类相关问题，均可直接联系该团队的开发支持专家。

最后，我们可以看到，容器化技术已经成为计算模型演化的一个开端，Kubernetes作为谷歌开源的Docker容器集群管理技术，在这场新的技术革命中扮演着重要的角色。Kubernetes正在被众多知名企业所采用，例如RedHat、VMware、CoreOS及腾讯等，因此，Kubernetes站在了容器新技术变革的浪潮之巅，将具有不可预估的发展前景和商业价值。

如果你是初级程序员，那么你有必要好好学习本书；如果你正在IT领域进行高级进阶修炼，那你也有必要阅读本书。无论是架构师、开发者、运维人员，还是对容器技术比较好奇的读者，本书都是一本不可多得的带你从入门向高级进阶的精品书，值得大家选择！

初瑞

中国移动业务支撑中心高级经理

自序

我不知道你是如何获得这本书的，可能是在百度头条、网络广告、朋友圈中听说本书后购买的，也可能是某一天逛书店时，这本书恰好神奇地翻落书架，出现在你面前，让你想起一千多年前那个意外得到《太公兵法》的传奇少年，你觉得这是冥冥之中上天的恩赐，于是果断带走。不管怎样，我相信多年以后，这本书仍然值得你回忆。

Kubernetes这个名字起源于古希腊，是舵手的意思，所以它的Logo既像一张渔网，又像一个罗盘。谷歌采用这个名字的一层深意就是：既然Docker把自己定位为驮着集装箱在大海上自在遨游的鲸鱼，那么谷歌就要以Kubernetes掌舵大航海时代的话语权，“捕获”和“指引”这条鲸鱼按照“主人”设定的路线巡游，确保谷歌倾力打造的新一代容器世界的宏伟蓝图顺利实现。

虽然Kubernetes自诞生至今才1年多，其第一个正式版本Kubernetes 1.0于2015年7月才发布，完全是个新生事物，但其影响力巨大，已经吸引了包括IBM、惠普、微软、红帽、Intel、VMware、CoreOS、Docker、Mesosphere、Mirantis等在内的众多业界巨头纷纷加入。红帽这个软件虚拟化领域的领导者之一，在容器技术方面已经完全“跟从”谷歌了，不仅把自家的第三代OpenShift产品的架构底层换成

了Docker+Kubernetes，还直接在其新一代容器操作系统Atomic内原生集成了Kubernetes。

Kubernetes是第一个将“一切以服务（Service）为中心，一切围绕服务运转”作为指导思想的创新型产品，它的功能和架构设计自始至终都遵循了这一指导思想，构建在Kubernetes上的系统不仅可以独立运行在物理机、虚拟机集群或者企业私有云上，也可以被托管在公有云中。Kubernetes方案的另一个亮点是自动化，在Kubernetes的解决方案中，一个服务可以自我扩展、自我诊断，并且容易升级，在收到服务扩容的请求后，Kubernetes会触发调度流程，最终在选定的目标节点上启动相应数量的服务实例副本，这些副本在启动成功后会自动加入负载均衡器中并生效，整个过程无须额外的人工操作。另外，Kubernetes会定时巡查每个服务的所有实例的可用性，确保服务实例的数量始终保持为预期的数量，当它发现某个实例不可用时，会自动重启该实例或者在其他节点重新调度、运行一个新实例，这样，一个复杂的过程无须人工干预即可全部自动化完成。试想一下，如果一个包括几十个节点且运行着几万个容器的复杂系统，其负载均衡、故障检测和故障修复等都需要人工介入进行处理，那将是多么难以想象。

通常我们会把Kubernetes看作Docker的上层架构，就好像Java与J2EE的关系一样：J2EE是以Java为基础的企业级软件架构，而Kubernetes则以Docker为基础打造了一个云计算时代的全新分布式系统架构。但Kubernetes与Docker之间还存在着更为复杂的关系，从表面上看，似乎Kubernetes离不开Docker，但实际上在Kubernetes的架构里，Docker只是其目前支持的两种底层容器技术之一，另一个容器技术则是Rocket，后者来源于CoreOS这个Docker昔日的“恋人”所推出的竞争产品。

Kubernetes同时支持这两种互相竞争的容器技术，这是有深刻的历史原因的。快速发展的Docker打败了谷歌曾经名噪一时的开源容器技术lmctfy，并迅速风靡世界。但是，作为一个已经对全球IT公司产生重要影响的技术，Docker背后的容器标准的制定注定不可能被任何一个公司私有控制，于是就有了后来引发危机的CoreOS与Docker分手事件，其导火索是CoreOS撇开了Docker，推出了与Docker相对抗的开源容器项目——Rocket，并动员一些知名IT公司成立委员会来试图主导容器技术的标准化，该分手事件愈演愈烈，最终导致CoreOS“傍上”谷歌一起宣布“叛逃”Docker阵营，共同发起了基于CoreOS+Rocket+Kubernetes的新项目Tectonic。这让当时的Docker阵营和Docker粉丝们无比担心Docker的命运，不管最终鹿死谁手，容器技术分裂态势的加剧对所有牵涉其中的人来说都没有好处，于是Linux基金会出面调和矛盾，双方都退让一步，最终的结果是Linux基金会于2015年6月宣布成立开放容器技术项目（Open Container Project），谷歌、CoreOS及Docker都加入了OCP项目。但通过查看OCP项目的成员名单，你会发现Docker在这个名单中只能算一个小角色了。OCP的成立最终结束了这场让无数人揪心的“战争”，Docker公司被迫放弃了自己的独家控制权。作为回报，Docker的容器格式被OCP采纳为新标准的基础，并且由Docker负责起草OCP草案规范的初稿文档，当然这个“标准起草者”的角色也不是那么容易担当的，Docker要提交自己的容器执行引擎的源码作为OCP项目的启动资源。

事到如今，我们再来回顾当初CoreOS与谷歌的叛逃事件，从表面上看，谷歌貌似是被诱拐“出柜”的，但局里人都明白，谷歌才是这一系列事件背后的主谋，其不仅为当年失败的lmctfy报了一箭之仇，还重新掌控了容器技术的未来。容器标准之战大捷之后，谷歌进一步扩大了联盟并提高了自身影响力。2015年7月，谷歌正式宣布加入

OpenStack阵营，其目标是确保Linux容器及关联的容器管理技术Kubernetes能够被OpenStack生态圈所容纳，并且成为OpenStack平台上与KVM虚拟机一样的一等公民。谷歌加入OpenStack意味着对数据中心控制平面的争夺已经结束，以容器为代表的形态与以虚拟化为代表的系统形态将会完美融合于OpenStack之上，并与软件定义网络和软件定义存储一起统治下一代数据中心。

谷歌凭借着几十年大规模容器使用的丰富经验，步步为营，先是祭出Kubernetes这个神器，然后又掌控了容器技术的制定标准，最后又入驻OpenStack阵营全力将Kubernetes扶上位，谷歌这个IT界的领导者和创新者再次王者归来。我们都明白，在IT世界里只有那些被大公司掌控和推广的，同时被业界众多巨头都认可和支持的新技术才能生存和壮大下去。Kubernetes就是当今IT界里符合要求且为数不多的热门技术之一，它的影响力可能长达十年，所以，我们每个IT人都有理由重视这门新技术。

谁能比别人领先一步掌握新技术，谁就在竞争中赢得了先机。惠普中国电信解决方案领域的资深专家团一起分工协作，并行研究，废寝忘食地合力撰写，在短短的5个月内完成了这部厚达500多页的Kubernetes权威指南。经过一年的高速发展，Kubernetes先后发布了1.1、1.2和1.3版本，每个版本都带来了大量的新特性，能够处理的应用场景也越来越丰富。本书遵循从入门到精通的学习路线，全书共分为六大章节，涵盖了入门、实践指南、架构原理、开发指南、高级案例、运维指南和源码分析等内容，内容详实、图文并茂，几乎囊括了Kubernetes 1.3版本的方方面面，无论是对于软件工程师、测试工程师、运维工程师、软件架构师、技术经理，还是对于资深IT人士来说，本书都极具参考价值。

吴治辉

惠普公司系统架构师

第1章 Kubernetes入门

1.1 Kubernetes是什么

Kubernetes是什么？

首先，它是一个全新的基于容器技术的分布式架构领先方案。这个方案虽然还很新，但它是谷歌十几年以来大规模应用容器技术的经验积累和升华的一个重要成果。确切地说，Kubernetes是谷歌严格保密十几年的秘密武器——Borg的一个开源版本。Borg是谷歌的一个久负盛名的内部使用的大规模集群管理系统，它基于容器技术，目的是实现资源管理的自动化，以及跨多个数据中心的资源利用率的最大化。十几年来，谷歌一直通过Borg系统管理着数量庞大的应用程序集群。由于谷歌员工都签署了保密协议，即便离职也不能泄露Borg的内部设计，所以外界一直无法了解关于它的更多信息。直到2015年4月，传闻许久的Borg论文伴随Kubernetes的高调宣传被谷歌首次公开，大家才得以了解它的更多内幕。正是由于站在Borg这个前辈的肩膀上，吸取了Borg过去十年间的经验与教训，所以Kubernetes一经开源就一鸣惊人，并迅速称霸了容器技术领域。

其次，如果我们的系统设计遵循了Kubernetes的设计思想，那么传统系统架构中那些和业务没有多大关系的底层代码或功能模块，都

可以立刻从我们的视线中消失，我们不必再费心于负载均衡器的选型和部署实施问题，不必再考虑引入或自己开发一个复杂的服务治理框架，不必再头疼于服务监控和故障处理模块的开发。总之，使用Kubernetes提供的解决方案，我们不仅节省了不少于30%的开发成本，同时可以将精力更加集中于业务本身，而且由于Kubernetes提供了强大的自动化机制，所以系统后期的运维难度和运维成本大幅度降低。

然后，Kubernetes是一个开放的开发平台。与J2EE不同，它不局限于任何一种语言，没有限定任何编程接口，所以不论是用Java、Go、C++还是用Python编写的服务，都可以毫无困难地映射为Kubernetes的Service，并通过标准的TCP通信协议进行交互。此外，由于Kubernetes平台对现有的编程语言、编程框架、中间件没有任何侵入性，因此现有的系统也很容易改造升级并迁移到Kubernetes平台上。

最后，Kubernetes是一个完备的分布式系统支撑平台。Kubernetes具有完备的集群管理能力，包括多层次的安全防护和准入机制、多租户应用支撑能力、透明的服务注册和服务发现机制、内建智能负载均衡器、强大的故障发现和自我修复能力、服务滚动升级和在线扩容能力、可扩展的资源自动调度机制，以及多粒度的资源配额管理能力。同时，Kubernetes提供了完善的管理工具，这些工具涵盖了包括开发、部署测试、运维监控在内的各个环节。因此，Kubernetes是一个全新的基于容器技术的分布式架构解决方案，并且是一个一站式的完备的分布式系统开发和支撑平台。

在正式开始本章的Hello World之旅之前，我们首先要学习Kubernetes的一些基本知识，这样我们才能理解Kubernetes提供的解决方案。

在Kubernetes中，Service（服务）是分布式集群架构的核心，一个Service对象拥有如下关键特征。

- 拥有一个唯一指定的名字（比如mysql-server）。
- 拥有一个虚拟IP（Cluster IP、Service IP或VIP）和端口号。
- 能够提供某种远程服务能力。
- 被映射到了提供这种服务能力的一组容器应用上。

Service的服务进程目前都基于Socket通信方式对外提供服务，比如Redis、Memcache、MySQL、Web Server，或者是实现了某个具体业务的一个特定的TCP Server进程。虽然一个Service通常由多个相关的服务进程来提供服务，每个服务进程都有一个独立的Endpoint（IP+Port）访问点，但Kubernetes能够让我们通过Service（虚拟Cluster IP+Service Port）连接到指定的Service上。有了Kubernetes内建的透明负载均衡和故障恢复机制，不管后端有多少服务进程，也不管某个服务进程是否会由于发生故障而重新部署到其他机器，都不会影响到我们对服务的正常调用。更重要的是这个Service本身一旦创建就不再变化，这意味着，在Kubernetes集群中，我们再也不用为了服务的IP地址变来变去的问题而头疼了。

容器提供了强大的隔离功能，所以有必要把为Service提供服务的这组进程放入容器中进行隔离。为此，Kubernetes设计了Pod对象，将每个服务进程包装到相应的Pod中，使其成为Pod中运行的一个容器（Container）。为了建立Service和Pod间的关联关系，Kubernetes首先给每个Pod贴上一个标签（Label），给运行MySQL的Pod贴上name=mysql标签，给运行PHP的Pod贴上name=php标签，然后给相应的Service定义标签选择器（Label Selector），比如MySQL Service的标签选择器的选择条件为name=mysql，意为该Service要作用于所有包含

name=mysql Label的Pod上。这样一来，就巧妙地解决了Service与Pod的关联问题。

说到Pod，我们这里先简单介绍其概念。首先，Pod运行在一个我们称之为节点（Node）的环境中，这个节点既可以是物理机，也可以是私有云或者公有云中的一个虚拟机，通常在一个节点上运行几百个Pod；其次，每个Pod里运行着一个特殊的被称之为Pause的容器，其他容器则为业务容器，这些业务容器共享Pause容器的网络栈和Volume挂载卷，因此它们之间的通信和数据交换更为高效，在设计时我们可以充分利用这一特性将一组密切相关的服务进程放入同一个Pod中；最后，需要注意的是，并不是每个Pod和它里面运行的容器都能“映射”到一个Service上，只有那些提供服务（无论是对内还是对外）的一组Pod才会被“映射”成一个服务。

在集群管理方面，Kubernetes将集群中的机器划分为一个Master节点和一群工作节点（Node）。其中，在Master节点上运行着集群管理相关的一组进程 kube-apiserver、kube-controller-manager 和 kube-scheduler，这些进程实现了整个集群的资源管理、Pod调度、弹性伸缩、安全控制、系统监控和纠错等管理功能，并且都是全自动完成的。Node作为集群中的工作节点，运行真正的应用程序，在Node上Kubernetes管理的最小运行单元是Pod。Node上运行着Kubernetes的 kubelet、kube-proxy服务进程，这些服务进程负责Pod的创建、启动、监控、重启、销毁，以及实现软件模式的负载均衡器。

最后，我们再来看看传统的IT系统中服务扩容和服务升级这两个难题，以及Kubernetes所提供的全新解决思路。服务的扩容涉及资源分配（选择哪个节点进行扩容）、实例部署和启动等环节，在一个复

杂的业务系统中，这两个问题基本上靠人工一步步操作才得以完成，费时费力又难以保证实施质量。

在Kubernetes集群中，你只需为需要扩容的Service关联的Pod创建一个Replication Controller（简称RC），则该Service的扩容以至于后来的Service升级等头疼问题都迎刃而解。在一个RC定义文件中包括以下3个关键信息。

- 目标Pod的定义。
- 目标Pod需要运行的副本数量（Replicas）。
- 要监控的目标Pod的标签（Label）。

在创建好RC（系统将自动创建好Pod）后，Kubernetes会通过RC中定义的Label筛选出对应的Pod实例并实时监控其状态和数量，如果实例数量少于定义的副本数量（Replicas），则会根据RC中定义的Pod模板来创建一个新的Pod，然后将此Pod调度到合适的Node上启动运行，直到Pod实例的数量达到预定目标。这个过程完全是自动化的，无须人工干预。有了RC，服务的扩容就变成了一个纯粹的简单数字游戏了，只要修改RC中的副本数量即可。后续的Service升级也将通过修改RC来自动完成。

以将在第2章介绍的PHP+Redis留言板应用为例，只要为PHP留言板程序（frontend）创建一个有3个副本的RC+Service，为Redis读写分离集群创建两个RC：写节点（redis-master）创建一个单副本的RC+Service，读节点（redis-slaver）创建一个有两个副本的RC+Service，就可以分分钟完成整个集群的搭建过程了，是不是很简单？

1.2 为什么要用Kubernetes

使用Kubernetes的理由很多，最根本的一个理由就是：IT从来都是一个由新技术驱动的行业。

Docker这个新兴的容器化技术当前已经被很多公司所采用，其从单机走向集群已成为必然，而云计算的蓬勃发展正在加速这一进程。Kubernetes作为当前唯一被业界广泛认可和看好的Docker分布式系统解决方案，可以预见，在未来几年内，会有大量的新系统选择它，不管这些系统是运行在企业本地服务器上还是被托管到公有云上。

使用了Kubernetes又会收获哪些好处呢？

首先，最直接的感受就是我们可以“轻装上阵”地开发复杂系统了。以前动不动就需要十几个人而且团队里需要不少技术达人一起分工协作才能设计实现和运维的分布式系统，在采用Kubernetes解决方案之后，只需一个精悍的小团队就能轻松应对。在这个团队里，一名架构师专注于系统中“服务组件”的提炼，几名开发工程师专注于业务代码的开发，一名系统兼运维工程师负责Kubernetes的部署和运维，从此再也不用“996”了，这并不是因为我们少做了什么，而是因为Kubernetes已经帮我们做了很多。

其次，使用Kubernetes就是在全面拥抱微服务架构。微服务架构的核心是将一个巨大的单体应用分解为很多小的互相连接的微服务，一个微服务背后可能有多个实例副本在支撑，副本的数量可能会随着系统的负荷变化而进行调整，内嵌的负载均衡器在这里发挥了重要作

用。微服务架构使得每个服务都可以由专门的开发团队来开发，开发者可以自由选择开发技术，这对于大规模团队来说很有价值，另外每个微服务独立开发、升级、扩展，因此系统具备很高的稳定性和快速迭代进化能力。谷歌、亚马逊、eBay、NetFlix等众多大型互联网公司都采用了微服务架构，此次谷歌更是将微服务架构的基础设施直接打包到Kubernetes解决方案中，让我们有机会直接应用微服务架构解决复杂业务系统的架构问题。

然后，我们的系统可以随时随地整体“搬迁”到公有云上。Kubernetes最初的目标就是运行在谷歌自家的公有云GCE中，未来会支持更多的公有云及基于OpenStack的私有云。同时，在Kubernetes的架构方案中，底层网络的细节完全被屏蔽，基于服务的Cluster IP甚至都无须我们改变运行期的配置文件，就能将系统从物理机环境中无缝迁移到公有云中，或者在服务高峰期将部分服务对应的Pod副本放入公有云中以提升系统的吞吐量，不仅节省了公司的硬件投入，还大大改善了客户体验。我们所熟知的铁道部的12306购票系统，在春节高峰期就租用了阿里云进行分流。

最后，Kubernetes系统架构具备了超强的横向扩容能力。对于互联网公司来说，用户规模就等价于资产，谁拥有更多的用户，谁就能在竞争中胜出，因此超强的横向扩容能力是互联网业务系统的关键指标之一。不用修改代码，一个Kubernetes集群即可从只包含几个Node的小集群平滑扩展到拥有上百个Node的大规模集群，我们利用Kubernetes提供的工具，甚至可以在线完成集群扩容。只要我们的微服务设计得好，结合硬件或者公有云资源的线性增加，系统就能够承受大量用户并发访问所带来的巨大压力。

1.3 从一个简单的例子开始

考虑到本书第1版中的**PHP+Redis**留言板的**Hello World**例子对于绝大多数刚接触**Kubernetes**的人来说比较复杂，难以顺利上手和实践，所以我们在此将这个例子替换成一个简单得多的**Java Web**应用，可以让新手快速上手和实践。

此**Java Web**应用的结构比较简单，是一个运行在**Tomcat**里的**Web App**，如图1.1所示，**JSP**页面通过**JDBC**直接访问**MySQL**数据库并展示数据。为了演示和简化的目的，只要程序正确连接到了数据库上，它就会自动完成对应的**Table**的创建与初始化数据的准备工作。所以，当我们通过浏览器访问此应用的时候，就会显示一个表格的页面，数据则来自数据库。

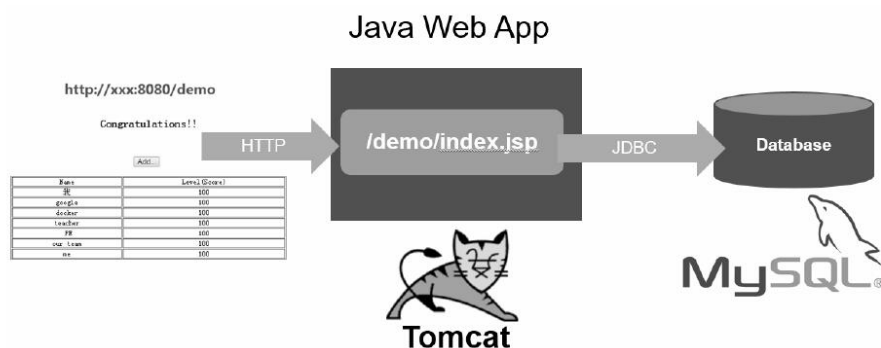


图1.1 Java Web应用的架构组成

此应用需要启动两个容器：Web App容器和MySQL容器，并且Web App容器需要访问MySQL容器。在**Docker**时代，假设我们

宿主主机上启动了这两个容器，则我们需要把MySQL容器的IP地址通过环境变量的方式注入Web App容器里；同时，需要将WebApp容器的8080端口映射到宿主机的8080端口，以便能在外部访问。在本章的这个例子里，我们看看在Kubernetes时代是如何完成这个目标的。

1.3.1 环境准备

首先，我们开始准备Kubernetes的安装和相关镜像下载，本书建议采用VirtualBox或者VMwareWorkstation在本机虚拟一个64位的CentOS 7虚拟机作为学习环境，虚拟机采用NAT的网络模式以便能够连接外网，然后按照以下步骤快速安装Kubernetes。

(1) 关闭CentOS自带的防火墙服务：

```
# systemctl disable firewalld
# systemctl stop firewalld
```

(2) 安装etcd和Kubernetes软件（会自动安装Docker软件）：

```
#yum install -y etcd kubernetes
```

(3) 安装好软件后，修改两个配置文件（其他配置文件使用系统默认的配置参数即可）。

- Docker配置文件为/etc/sysconfig/docker，其中OPTIONS的内容设置为：

```
OPTIONS='--selinux-enabled=false --insecure-registry
gcr.io'
```

- Kubernetes apiserver 配置文件为 /etc/kubernetes/apiserver , 把 --admission_control 参数中的 ServiceAccount 删除。

(4) 按顺序启动所有的服务:

```
#systemctl start etcd
#systemctl start docker
#systemctl start kube-apiserver
#systemctl start kube-controller-manager
#systemctl start kube-scheduler
#systemctl start kubelet
#systemctl start kube-proxy
```

至此, 一个单机版的Kubernetes集群环境就安装启动完成了。

接下来, 我们可以在这个单机版的Kubernetes集群中上手练习了。

注: 本书示例中的 Docker 镜像下载地址为 <https://hub.docker.com/u/kubeguide/>。

1.3.2 启动MySQL服务

首先为MySQL服务创建一个RC定义文件：`mysql-rc.yaml`，下面给出了该文件的完整内容和解释，如图1.2所示。

<code>apiVersion: v1</code>	
<code>kind: ReplicationController</code>	副本控制器 RC
<code>metadata:</code>	
<code>name: mysql</code>	RC 的名称，全局唯一
<code>spec:</code>	
<code>replicas: 1</code>	Pod 副本期待数量
<code>selector:</code>	
<code>app: mysql</code>	符合目标的 Pod 拥有此标签
<code>template:</code>	根据此模板创建 Pod 的副本（实例）
<code>metadata:</code>	
<code>labels:</code>	
<code>app: mysql</code>	Pod 副本拥有的标签，对应 RC 的 Selector
<code>spec:</code>	
<code>containers:</code>	Pod 内容器的定义部分
<code>- name: mysql</code>	容器的名称
<code>image: mysql</code>	容器对应的 Docker Image
<code>ports:</code>	
<code>- containerPort: 3306</code>	容器暴露的端口号
<code>env:</code>	注入到容器内的环境变量
<code>- name: MYSQL_ROOT_PASSWORD</code>	
<code>value: "123456"</code>	

图1.2 RC的定义和解说图

`yaml`定义文件中的`kind`属性，用来表明此资源对象的类型，比如这里的值为“`ReplicationController`”，表示这是一个RC；`spec`一节中是RC的相关属性定义，比如`spec.selector`是RC的Pod标签（Label）选择器，即监控和管理拥有这些标签的Pod实例，确保当前集群上始终有且仅有`replicas`个Pod实例在运行，这里我们设置`replicas=1`表示只能运行一个MySQL Pod实例。当集群中运行的Pod数量小于`replicas`时，RC会根据`spec.template`一节中定义的Pod模板来生成一个新的Pod实例，

`spec.template.metadata.labels`指定了该Pod的标签，需要特别注意的是：这里的labels必须匹配之前的`spec.selector`，否则此RC每次创建了一个无法匹配Label的Pod，就会不停地尝试创建新的Pod，最终陷入“只为他人做嫁衣”的悲惨世界中，永无翻身之时。

创建好`redis-master-controller.yaml`文件以后，为了将它发布到Kubernetes集群中，我们在Master节点执行命令：

```
# kubectl create -f mysql-rc.yaml
replicationcontroller "mysql" created
```

接下来，我们用`kubectl`命令查看刚刚创建的RC：

```
# kubectl get rc
```

NAME	DESIRED	CURRENT	AGE
mysql	1	1	7m

查看Pod的创建情况时，可以运行下面的命令：

```
# kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
mysql-c95jc	1/1	Running	0	9m

我们看到一个名为`mysql-xxxxx`的Pod实例，这是Kubernetes根据`mysql`这个RC的定义自动创建的Pod。由于Pod的调度和创建需要花费一定的时间，比如需要一定的时间来确定调度到哪个节点上，以及下载Pod里容器的镜像需要一段时间，所以一开始我们看到Pod的状态将

显示为 **Pending** 。当 **Pod** 成功创建完成以后，状态最终会被更新为 **Running** 。

我们通过 `docker ps` 指令查看正在运行的容器，发现提供 **MySQL** 服务的 **Pod** 容器已经创建并正常运行了，此外，你会发现 **MySQL Pod** 对应的容器还多创建了一个来自谷歌的 `pause` 容器，这就是 **Pod** 的“根容器”，详见 1.4.3 节的说明。

```
# docker ps |grep mysql
72ca992535b4      mysql
"docker-entrypoint.sh" 12 minutes ago    Up 12 minutes
k8s_mysql.86dc506e_mysql-c95jc_default_511d6705-5051-11e6-
a9d8-000c29ed42c1_9f89d0b4
76c1790aad27      gcr.io/google_containers/pause-
amd64:3.0          "/pause"          12 minutes
ago                Up 12 minutes
k8s_POD.16b20365_mysql-c95jc_default_511d6705-5051-11e6-
a9d8-000c29ed42c1_28520aba
```

最后，我们创建一个与之关联的 **Kubernetes Service**——**MySQL** 的定义文件（文件名为 `mysql-svc.yaml`），完整的内容和解释如图 1.3 所示。

```
apiVersion: v1
kind: Service      表明是 Kubernetes Service
metadata:
  name: mysql      Service 的全局唯一名称
spec:
  ports:
    - port: 3306   Service 提供服务的端口号
  selector:        Service 对应的 Pod 拥有这里定义的标签
    app: mysql
```

图1.3 Service的定义和解说图

其中，`metadata.name`是Service的服务名（`ServiceName`）；`port`属性则定义了Service的虚端口；`spec.selector`确定了哪些Pod副本（实例）对应到本服务。类似地，我们通过`kubectl create`命令创建Service对象。

运行`kubectl`命令，创建service:

```
# kubectl create -f mysql-svc.yaml
service "mysql" created
```

运行`kubectl`命令，可以查看到刚刚创建的service:

```
# kubectl get svc
```

	NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)
AGE				
	mysql	169.169.253.143	<none>	3306/TCP
48s				

注意到MySQL服务被分配了一个值为169.169.253.143的ClusterIP地址，这是一个虚地址，随后，Kubernetes集群中其他新创建的Pod就可以通过Service的ClusterIP+端口号6379来连接和访问它了。

在通常情况下，ClusterIP是在Service创建后由Kubernetes系统自动分配的，其他Pod无法预先知道某个Service的ClusterIP地址，因此需要一个服务发现机制来找到这个服务。为此，最初的时候，Kubernetes巧妙地使用了Linux环境变量（Environment Variable）来解决这个问

题，后面会详细说明其机制。现在我们只需知道，根据Service的唯一名字，容器可以从环境变量中获取到Service对应的Cluster IP地址和端口，从而发起TCP/IP连接请求了。

1.3.3 启动Tomcat应用

上面我们定义和启动了MySQL服务，接下来我们采用同样的步骤，完成Tomcat应用的启动过程。首先，创建对应的RC文件myweb-rc.yaml，内容如下：

```
kind: ReplicationController
metadata:
  name: myweb
spec:
  replicas: 5
  selector:
    app: myweb
  template:
    metadata:
      labels:
        app: myweb
    spec:
      containers:
        - name: myweb
          image: kubeguide/tomcat-app:v1
          ports:
            - containerPort: 8080
          env:
```

```
- name: MYSQL_SERVICE_HOST
  value: 'mysql'
- name: MYSQL_SERVICE_PORT
  value: '3306'
```

注意到上面 RC 对应的 Tomcat 容器里引用了 `MYSQL_SERVICE_HOST=mysql` 这个环境变量，而“mysql”恰好是我们之前定义的MySQL服务的服务名，运行下面的命令，完成RC的创建和验证工作：

```
#kubectl create -f myweb-rc.yaml
replicationcontroller "myweb" created
```

```
#kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
mysql-c95jc	1/1	Running	0	2h
myweb-g9pmm	1/1	Running	0	3s

最后，创建对应的Service。以下是完整的yaml定义文件（myweb-svc.yaml）：

```
apiVersion: v1
kind: Service
metadata:
  name: myweb
spec:
  type: NodePort
```



```
ports:
  - port: 8080
    nodePort: 30001
selector:
  app: myweb
```

注意`type=NodePort`和`nodePort=30001`的两个属性，表明此Service开启了NodePort方式的外网访问模式，在Kubernetes集群之外，比如在本机的浏览器里，可以通过30001这个端口访问myweb（对应到8080的虚端口上）。

运行`kubectl create`命令进行创建：

```
# kubectl create -f myweb-svc.yaml
```

You have exposed your service on an external port on all nodes in your cluster. If you want to expose this service to the external internet, you may need to set up firewall rules for the service port(s) (tcp:30001) to serve traffic.

See <http://releases.k8s.io/release-1.3/docs/user-guide/services-firewalls.md> for more details.

```
service "myweb" created
```

我们看到上面有提示信息，意思是需要把30001这个端口在防火墙上打开，以便外部的访问能穿过防火墙。

运行`kubectl`命令，查看创建的Service：


# kubectl get services				
	NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)
AGE				
	mysql	169.169.253.143	<none>	3306/TCP
44m				
	myweb	169.169.149.215	<nodes>	8080/TCP
4m				
	kubernetes	169.169.0.1		<none>
443/TCP		16d		

至此，我们的第1个Kubernetes例子搭建完成了，在下一节中我们验证结果。

1.3.4 通过浏览器访问网页

经过上面的几个步骤，我们终于成功实现了Kubernetes上第1个例子的部署搭建工作。现在一起来见证成果吧，在你的笔记本上打开浏览器，输入`http://虚拟机IP: 30001/demo/`。

比如虚拟机IP为192.168.18.131（可以通过`#ip a`命令进行查询），在浏览器里输入地址`http://192.168.18.131: 30001/demo/`后，看到了如图1.4所示的网页界面，那么恭喜你，之前的努力没有白费，顺利闯关成功！



The screenshot shows a web application interface. At the top, it says "Congratulations!!". Below this is a button labeled "Add...". Underneath the button is a table with two columns: "Name" and "Level (Score)". The table contains six rows of data.

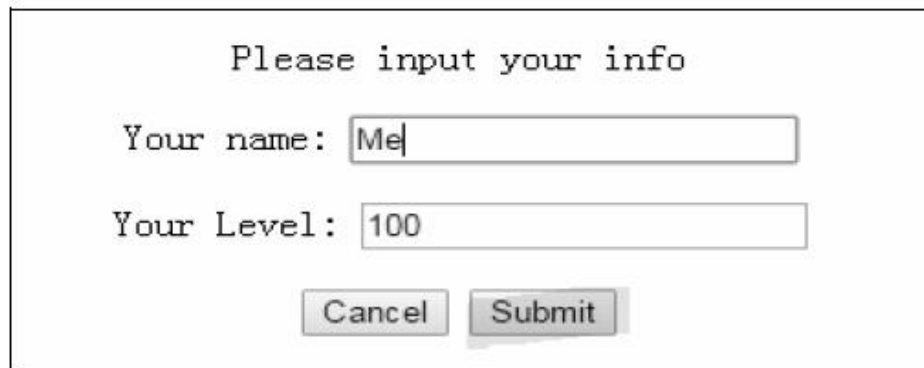
Name	Level (Score)
google	100
docker	100
teacher	100
PE	100
our team	100
me	100

图1.4 通过浏览器访问Tomcat应用

如果看不到这个网页，那么可能有几个原因：比如防火墙的问题，无法访问30001端口，或者因为你是通过代理上网的，浏览器错把虚拟机的IP地址当成远程地址了。可以在虚拟机上直接运行`curl localhost: 30001`来验证此端口是否能被访问，如果还是不能访问，那么这肯定不是机器的问题.....

接下来可以尝试单击“Add...”按钮添加一条记录并提交，如图1.5所示，提交以后，数据就被写入MySQL数据库中了。

至此，我们终于完成了Kubernetes上的Tomcat例子，这个例子并不是很复杂。我们也看到，相对于传统的分布式应用的部署方式，在Kubernetes之上我们仅仅通过一些很容易理解的配置文件和相关的简单命令就完成了对整个集群的部署，这让我们惊诧于Kubernetes的创新和强大。



Please input your info

Your name:

Your Level:

图1.5 在留言板网页添加新的留言

下一节，我们将开始对Kubernetes中的基本概念和术语进行全面学习，在这之前，读者可以继续研究下这个例子里的一些拓展内容，如下所述。

- 研究RC、Service等文件的格式。
- 熟悉kubectl的子命令。
- 手工停止某个Service对应的容器进程，然后观察有什么现象发生。
- 修改RC文件，改变副本数量，重新发布，观察结果。

1.4 Kubernetes基本概念和术语

Kubernetes中的大部分概念如Node、Pod、ReplicationController、Service等都可以看作一种“资源对象”，几乎所有的资源对象都可以通过Kubernetes提供的kubectl工具（或者API编程调用）执行增、删、改、查等操作并将其保存在etcd中持久化存储。从这个角度来看，Kubernetes其实是一个高度自动化的资源控制系统，它通过跟踪对比etcd库里保存的“资源期望状态”与当前环境中的“实际资源状态”的差异来实现自动控制和自动纠错的高级功能。

1.4.1 Master

Kubernetes里的Master指的是集群控制节点，每个Kubernetes集群里需要有一个Master节点来负责整个集群的管理和控制，基本上Kubernetes所有的控制命令都是发给它，它来负责具体的执行过程，我们后面所有执行的命令基本都是在Master节点上运行的。Master节点通常会占据一个独立的X86服务器（或者一个虚拟机），一个主要的原因是它太重要了，它是整个集群的“首脑”，如果它宕机或者不可用，那么我们所有的控制命令都将失效。

Master节点上运行着以下一组关键进程。

- Kubernetes API Server（kube-apiserver），提供了HTTP Rest接口的关键服务进程，是Kubernetes里所有资源的增、删、改、查等操作的唯一入口，也是集群控制的入口进程。
- Kubernetes Controller Manager（kube-controller-manager），Kubernetes里所有资源对象的自动化控制中心，可以理解为资源对象的“大总管”。
- Kubernetes Scheduler（kube-scheduler），负责资源调度（Pod调度）的进程，相当于公交公司的“调度室”。

其实Master节点上往往还启动了一个etcd Server进程，因为Kubernetes里的所有资源对象的数据全部是保存在etcd中的。

1.4.2 Node

除了Master，Kubernetes集群中的其他机器被称为Node节点，在较早的版本中也被称为Minion。与Master一样，Node节点可以是一台物理主机，也可以是一台虚拟机。Node节点才是Kubernetes集群中的工作负载节点，每个Node都会被Master分配一些工作负载（Docker容器），当某个Node宕机时，其上的工作负载会被Master自动转移到其他节点上去。

每个Node节点上都运行着以下一组关键进程。

- **kubelet**: 负责Pod对应的容器的创建、启停等任务，同时与Master节点密切协作，实现集群管理的基本功能。
- **kube-proxy**: 实现Kubernetes Service的通信与负载均衡机制的重要组件。
- **Docker Engine (docker)**: Docker引擎，负责本机的容器创建和管理工作。

Node节点可以在运行期间动态增加到Kubernetes集群中，前提是这个节点上已经正确安装、配置和启动了上述关键进程，在默认情况下kubelet会向Master注册自己，这也是Kubernetes推荐的Node管理方式。一旦Node被纳入集群管理范围，kubelet进程就会定时向Master节点汇报自身的情报，例如操作系统、Docker版本、机器的CPU和内存情况，以及之前有哪些Pod在运行等，这样Master可以获知每个Node的资源使用情况，并实现高效均衡的资源调度策略。而某个Node超过指

定时间不上报信息时，会被Master判定为“失联”，Node的状态被标记为不可用（Not Ready），随后Master会触发“工作负载大转移”的自动流程。

我们可以执行下述命令查看集群中有多少个Node:

# kubectl get nodes			
NAME	STATUS	AGE	
kubernetes-minion1	Ready	2d	

然后，通过kubectl describe node<node_name>来查看某个Node的详细信息:

\$ kubectl describe node kubernetes-minion1			
Name:	k8s-node-1		
Labels:	beta.kubernetes.io/arch=amd64		
	beta.kubernetes.io/os=linux		
	kubernetes.io/hostname=k8s-node-1		
Taints:	<none>		
CreationTimestamp:	Wed, 06 Jul 2016 11:46:41 +0800		
Phase:			
Conditions:			
Type		Status	LastHeartbeatTime
xxxx	OutOfDisk	False	Sat, 09 Jul 2016
08:17:39 +0800	Wed	
	MemoryPressure	False	Sat, 09 Jul 2016 08:17:39

+0800 Wed
Ready True Sat, 09 Jul 2016

08:17:39 +0800 Wed

Addresses: 192.168.18.131, 192.168.18.131

Capacity:

alpha.kubernetes.io/nvidia-gpu: 0

cpu: 4

memory: 1868692Ki

Pods: 110

Allocatable:

alpha.kubernetes.io/nvidia-gpu: 0

cpu: 4

memory: 1868692Ki

Pods: 110

System Info:

Machine ID:

6e4e2af2afeb42b9aac47d866aa56ca0

System UUID: 564D63D3-9664-3393-

A3DC-9CD424ED42C1

Boot ID: b0c34f9f-76ab-478e-9771-

bd4fe6e98880

Kernel Version: 3.10.0-327.22.2.el7.x86_64

OS Image: CentOS Linux 7 (Core)

Operating System: linux

Architecture: amd64

Container Runtime Version: docker://1.11.2

Kubelet Version: v1.3.0

```

    Kube-Proxy Version:                v1.3.0
    ExternalID:                        k8s-node-1
    Non-terminated Pods:                (1 in total)

      Namespace                                Name
CPU Requests    CPU Limits      Memory xxx
-----
-----
      kube-system                                kube-dns-v11-wxdhf
310m (7%)       310m (7%)       170Mi (9%)
    Allocated resources:
      (Total limits may be over 100 percent, i.e.,
overcommitted. More info:

      CPU Requests                CPU Limits      Memory
Requests    Memory Limits
-----
-----
      310m (7%)  310m (7%)       170Mi (9%)       170Mi (9%)
    No events.

```

上述命令展示了Node的如下关键信息。

- Node基本信息：名称、标签、创建时间等。
- Node当前的运行状态，Node启动以后会做一系列的自检工作，比如磁盘是否满了，如果满了就标注OutOfDisk=True，否则继续检查内存是否不足（MemoryPressure=True），最后一切正常，就切换为Ready状态（Ready=True），这种情况表示Node处于健康状态，可以在其上创建新的Pod。

- Node的主机地址与主机名。
- Node上的资源总量：描述Node可用的系统资源，包括CPU、内存数量、最大可调度Pod数量等，注意到目前Kubernetes已经实验性地支持GPU资源分配了（alpha.kubernetes.io/nvidia-gpu=0）。
- Node可分配资源量：描述Node当前可用于分配的资源量。
- 主机系统信息：包括主机的唯一标识UUID、Linux kernel版本号、操作系统类型与版本、Kubernetes版本号、kubelet与kube-proxy的版本号等。
- 当前正在运行的Pod列表概要信息。
- 已分配的资源使用概要信息，例如资源申请的最低、最大允许使用量占系统总量的百分比。
- Node相关的Event信息。

1.4.3 Pod

Pod是Kubernetes的最重要也最基本的概念，如图1.6所示是Pod的组成示意图，我们看到每个Pod都有一个特殊的被称为“根容器”的Pause容器。Pause容器对应的镜像属于Kubernetes平台的一部分，除了Pause容器，每个Pod还包含一个或多个紧密相关的用户业务容器。

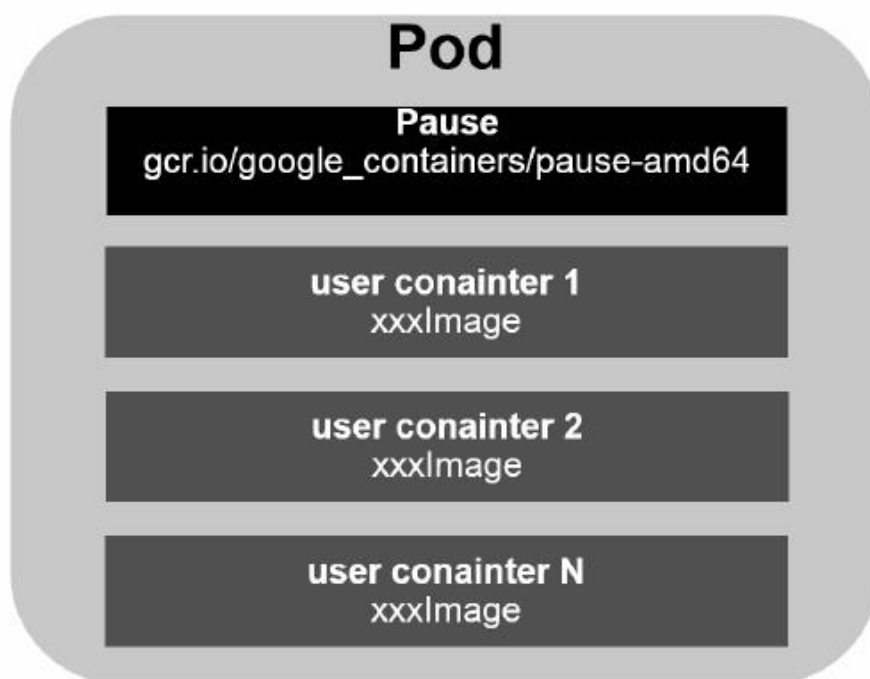


图1.6 Pod的组成与容器的关系

为什么Kubernetes会设计出一个全新的Pod的概念并且Pod有这样特殊的组成结构？

原因之一：在一组容器作为一个单元的情况下，我们难以对“整体”简单地进行判断及有效地进行行动。比如，一个容器死亡了，此时算是整体死亡么？是N/M的死亡率么？引入业务无关并且不易死亡的Pause容器作为Pod的根容器，以它的状态代表整个容器组的状态，就简单、巧妙地解决了这个难题。

原因之二：Pod里的多个业务容器共享Pause容器的IP，共享Pause容器挂接的Volume，这样既简化了密切关联的业务容器之间的通信问题，也很好解决了它们之间的文件共享问题。

Kubernetes为每个Pod都分配了唯一的IP地址，称之为Pod IP，一个Pod里的多个容器共享Pod IP地址。Kubernetes要求底层网络支持集群内任意两个Pod之间的TCP/IP直接通信，这通常采用虚拟二层网络技术来实现，例如Flannel、Openvswitch等，因此我们需要牢记一点：在Kubernetes里，一个Pod里的容器与另外主机上的Pod容器能够直接通信。

Pod其实有两种类型：普通的Pod及静态Pod（static Pod），后者比较特殊，它并不存放在Kubernetes的etcd存储里，而是存放在某个具体的Node上的一个具体文件中，并且只在此Node上启动运行。而普通的Pod一旦被创建，就会被放入到etcd中存储，随后会被Kubernetes Master调度到某个具体的Node上并进行绑定（Binding），随后该Pod被对应的Node上的kubelet进程实例化成一组相关的Docker容器并启动起来。在默认情况下，当Pod里的某个容器停止时，Kubernetes会自动检测到这个问题并且重新启动这个Pod（重启Pod里的所有容器），如果Pod所在的Node宕机，则会将这个Node上的所有Pod重新调度到其他节点上。Pod、容器与Node的关系如图1.7所示。

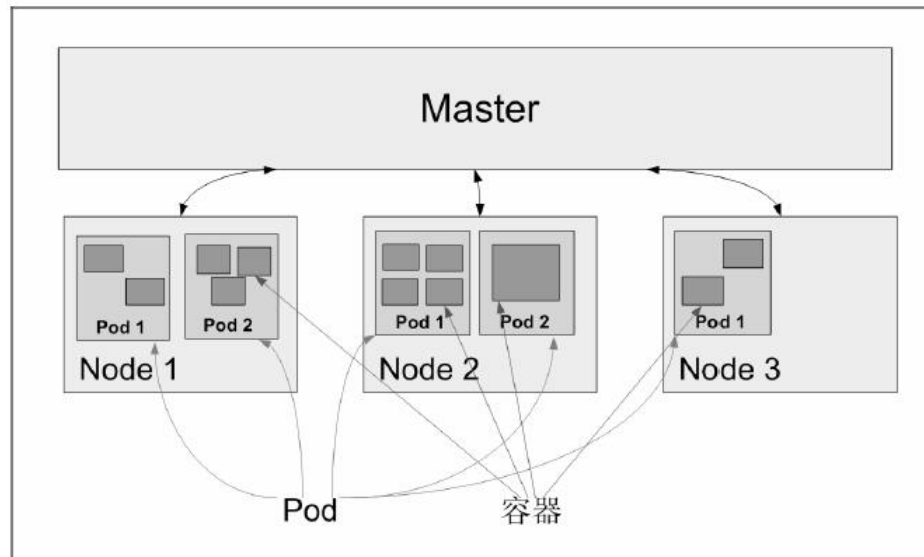


图1.7 Pod、容器与Node的关系

Kubernetes里的所有资源对象都可以采用yaml或者JSON格式的文件来定义或描述，下面是我们在之前Hello World例子里用到的myweb这个Pod的资源定义文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: myweb
  labels:
    name: myweb
spec:
  containers:
  - name: myweb
    image: kubeguide/tomcat-app:v1
    ports:
```

```
- containerPort: 8080
env:
- name: MYSQL_SERVICE_HOST
  value: 'mysql'
- name: MYSQL_SERVICE_PORT
  value: '3306'
```

Kind为Pod表明这是一个Pod的定义，metadata里的name属性为Pod的名字，metadata里还能定义资源对象的标签（Label），这里声明myweb拥有一个name=myweb的标签（Label）。Pod里所包含的容器组的定义则在spec一节中声明，这里定义了一个名字为myweb、对应镜像为kubeguide/tomcat-app:v1的容器，该容器注入了名为MYSQL_SERVICE_HOST='mysql'和MYSQL_SERVICE_PORT='3306'的环境变量（env关键字），并且在8080端口（containerPort）上启动容器进程。Pod的IP加上这里的容器端口（containerPort），就组成了一个新的概念——Endpoint，它代表着此Pod里的一个服务进程的对外通信地址。一个Pod也存在着具有多个Endpoint的情况，比如当我们把Tomcat定义为一个Pod的时候，可以对外暴露管理端口与服务端口这两个Endpoint。

我们所熟悉的Docker Volume在Kubernetes里也有对应的概念——Pod Volume，后者有一些扩展，比如可以用分布式文件系统GlusterFS实现后端存储功能；Pod Volume是定义在Pod之上，然后被各个容器挂载到自己的文件系统上的。

这里顺便提一下Kubernetes的Event概念，Event是一个事件的记录，记录了事件的最早产生时间、最后重现时间、重复次数、发起

者、类型，以及导致此事件的原因等众多信息。**Event**通常会关联到某个具体的资源对象上，是排查故障的重要参考信息，之前我们看到**Node**的描述信息包括了**Event**，而**Pod**同样有**Event**记录，当我们发现某个**Pod**迟迟无法创建时，可以用**kubectl describe pod xxxx**来查看它的描述信息，用来定位问题的原因，比如下面这个**Event**记录信息表明**Pod**里的一个容器被探针检测为失败一次：

Events:				
FirstSeen	LastSeen	Count	From	
SubobjectPath	Type	Reason	Message	
-----	-----	-----	-----	----
-----	-----	-----	-----	
10h	12m	32	{kubelet	
k8s-node-1}	spec.containers{kube2sky}		Warning	
Unhealthy	Liveness probe	failed:	Get	
http://172.17.1.2:8080/healthz: net/http: request canceled				
(Client.Timeout exceeded while awaiting headers)				

每个**Pod**都可以对其能使用的服务器上的计算资源设置限额，当前可以设置限额的计算资源有**CPU**与**Memory**两种，其中**CPU**的资源单位为**CPU（Core）**的数量，是一个绝对值而非相对值。

一个**CPU**的配额对于绝大多数容器来说是相当大的一个资源配额了，所以，在**Kubernetes**里，通常以千分之一的**CPU**配额为最小单位，用**m**来表示。通常一个容器的**CPU**配额被定义为**100~300m**，即占用**0.1~0.3**个**CPU**。由于**CPU**配额是一个绝对值，所以无论在拥有一个**Core**的机器上，还是在拥有**48**个**Core**的机器上，**100m**这个配额所代表

的CPU的使用量都是一样的。与CPU配额类似，Memory配额也是一个绝对值，它的单位是内存字节数。

在Kubernetes里，一个计算资源进行配额限定需要设定以下两个参数。

- **Requests**: 该资源的最小申请量，系统必须满足要求。
- **Limits**: 该资源最大允许使用的量，不能被突破，当容器试图使用超过这个量的资源时，可能会被Kubernetes Kill并重启。

通常我们会把Request设置为一个比较小的数值，符合容器平时的工作负载情况下的资源需求，而把Limit设置为峰值负载情况下资源占用的最大量。比如下面这段定义，表明MySQL容器申请最少0.25个CPU及64MiB内存，在运行过程中MySQL容器所能使用的资源配额为0.5个CPU及128MiB内存：

```
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

本节最后，笔者给出Pod及Pod周边对象的示意图作为总结，如图1.8所示，后面部分还会涉及这张图里的对象和概念，以进一步加强理解。

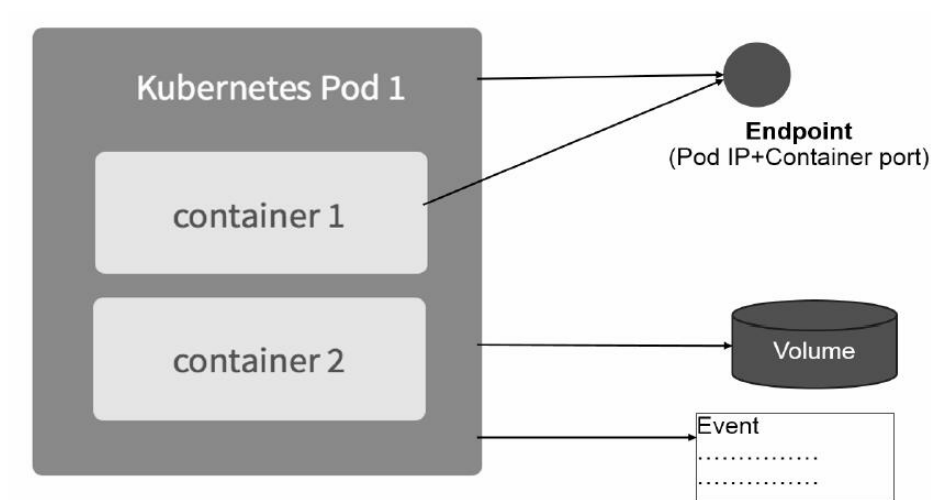


图1.8 Pod及周边对象

1.4.4 Label（标签）

Label是Kubernetes系统中另外一个核心概念。一个Label是一个key=value的键值对，其中key与value由用户自己指定。Label可以附加到各种资源对象上，例如Node、Pod、Service、RC等，一个资源对象可以定义任意数量的Label，同一个Label也可以被添加到任意数量的资源对象上去，Label通常在资源对象定义时确定，也可以在对象创建后动态添加或者删除。

我们可以通过给指定的资源对象捆绑一个或多个不同的Label来实现多维度的资源分组管理功能，以便于灵活、方便地进行资源分配、调度、配置、部署等管理工作。例如：部署不同版本的应用到不同的环境中；或者监控和分析应用（日志记录、监控、告警）等。一些常用的Label示例如下。

- 版本标签: “release”: “stable”, “release”: “canary”...
- 环境标签: “environment”: “dev”, “environment”: “qa”, “environment”: “production”
- 架构标签: “tier”: “frontend”, “tier”: “backend”, “tier”: “middleware”
- 分区标签: “partition”: “customerA”, “partition”: “customerB”...
- 质量管控标签: “track”: “daily”, “track”: “weekly”

Label相当于我们熟悉的“标签”，给某个资源对象定义一个Label，就相当于给它打了一个标签，随后可以通过Label Selector（标签选择器）查询和筛选拥有某些Label的资源对象，Kubernetes通过这种方式实现了类似SQL的简单又通用的对象查询机制。

Label Selector可以被类比为SQL语句中的where查询条件，例如，name=redis-slave 这个 Label Selector 作用于 Pod 时，可以被类比为select*from pod where pod's name='redis-slave'这样的语句。当前有两种Label Selector的表达式：基于等式的（Equality-based）和基于集合的（Set-based），前者采用“等式类”的表达式匹配标签，下面是一些具体的例子。

- name=redis-slave：匹配所有具有标签name=redis-slave的资源对象。
- env !=production：匹配所有不具有标签env=production的资源对象，比如env=test就是满足此条件的标签之一。

而后者则使用集合操作的表达式匹配标签，下面是一些具体的例子。

- name in (redis-master , redis-slave) ：匹配所有具有标签name=redis-master或者name=redis-slave的资源对象。
- name notin (php-frontend) ：匹配所有不具有标签name=php-frontend的资源对象。

可以通过多个Label Selector表达式的组合实现复杂的条件选择，多个表达式之间用“，”进行分隔即可，几个条件之间是“AND”的关系，即同时满足多个条件，比如下面的例子：

```
name=redis-slave,env!=production  
name notin (php-frontend),env!=production
```

Label Selector在Kubernetes中的重要使用场景有以下几处。

- kube-controller进程通过资源对象RC上定义的Label Selector来筛选要监控的Pod副本的数量，从而实现Pod副本的数量始终符合预期设定的全自动控制流程。
- kube-proxy进程通过Service的Label Selector来选择对应的Pod，自动建立起每个Service到对应Pod的请求转发路由表，从而实现Service的智能负载均衡机制。
- 通过对某些Node定义特定的Label，并且在Pod定义文件中使用NodeSelector这种标签调度策略， kube-scheduler进程可以实现Pod“定向调度”的特性。

在前面的留言板例子中，我们只使用了一个name=XXX的Label Selector。让我们看一个更复杂的例子。假设为Pod定义了3个Label: release、env和role，不同的Pod定义了不同的Label值，如图1.9所示，如果我们设置了“role=frontend”的LabelSelector，则会选取到Node 1和Node2上的Pod。

而设置“release=beta”的LabelSelector，则会选取到Node 2和Node3上的Pod，如图1.10所示。

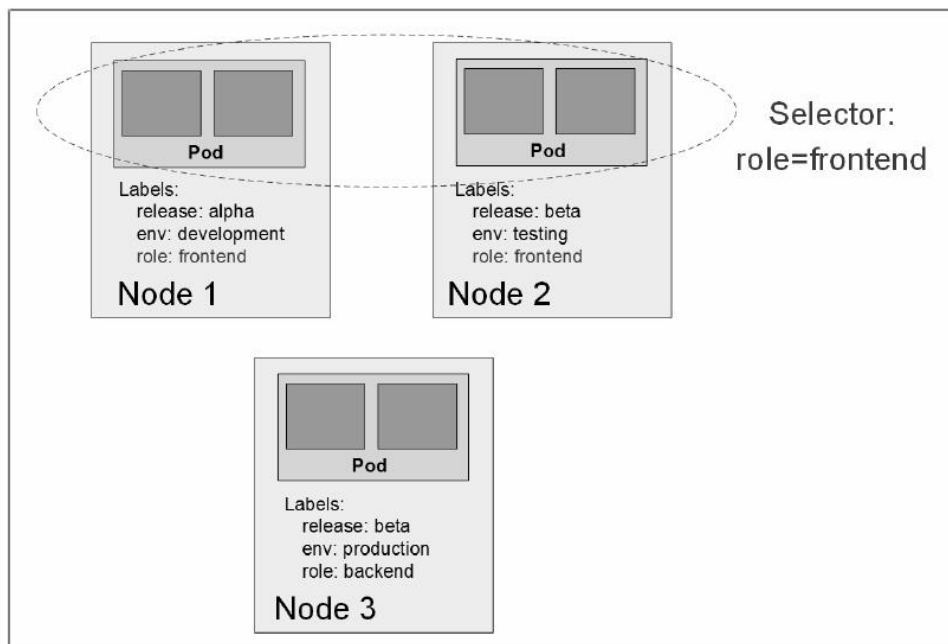


图1.9 Label Selector的作用范围1

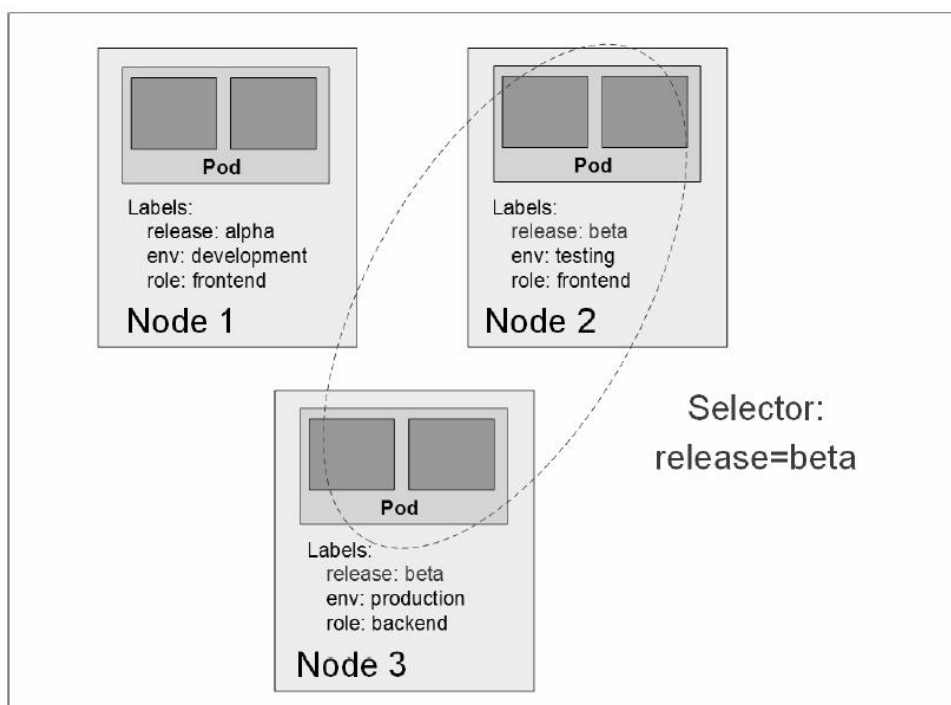


图1.10 Label Selector的作用范围2

总结：使用Label可以给对象创建多组标签，Label和LabelSelector共同构成了Kubernetes系统中最核心的应用模型，使得被管理对象能够被精细地分组管理，同时实现了整个集群的高可用性。

1.4.5 Replication Controller (RC)

之前已经对Replication Controller（以后简称RC）的定义和作用做了一些说明，本节对RC的概念进行深入描述。

RC是Kubernetes系统中的核心概念之一，简单来说，它其实是定义了一个期望的场景，即声明某种Pod的副本数量在任意时刻都符合某个预期值，所以RC的定义包括如下几个部分。

- Pod期待的副本数（replicas）。
- 用于筛选目标Pod的Label Selector。
- 当Pod的副本数量小于预期数量的时候，用于创建新Pod的Pod模板（template）。

下面是一个完整的RC定义的例子，即确保拥有tier=frontend标签的这个Pod（运行Tomcat容器）在整个Kubernetes集群中始终只有一个副本。

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
spec:
  replicas: 1
  selector:
```



```
    tier: frontend
template:
  metadata:
    labels:
      app: app-demo
      tier: frontend
  spec:
    containers:
      - name: tomcat-demo
        image: tomcat
        imagePullPolicy: IfNotPresent
        env:
          - name: GET_HOSTS_FROM
            value: dns
        ports:
          - containerPort: 80
```

当我们定义了一个RC并提交到Kubernetes集群中以后，Master节点上的Controller Manager组件就得到通知，定期巡检系统中当前存活的目标Pod，并确保目标Pod实例的数量刚好等于此RC的期望值，如果有过多的Pod副本在运行，系统就会停掉一些Pod，否则系统就会再自动创建一些Pod。可以说，通过RC，Kubernetes实现了用户应用集群的高可用性，并且大大减少了系统管理员在传统IT环境中需要完成的许多手工运维工作（如主机监控脚本、应用监控脚本、故障恢复脚本等）。

下面我们以3个Node节点的集群为例，说明Kubernetes如何通过RC来实现Pod副本数量自动控制的机制。假如我们的RC里定义redis-slave这个Pod需要保持3个副本，系统将可能在其中的两个Node上创建Pod。图1.11描述了在两个Node上创建redis-slave Pod的情形。

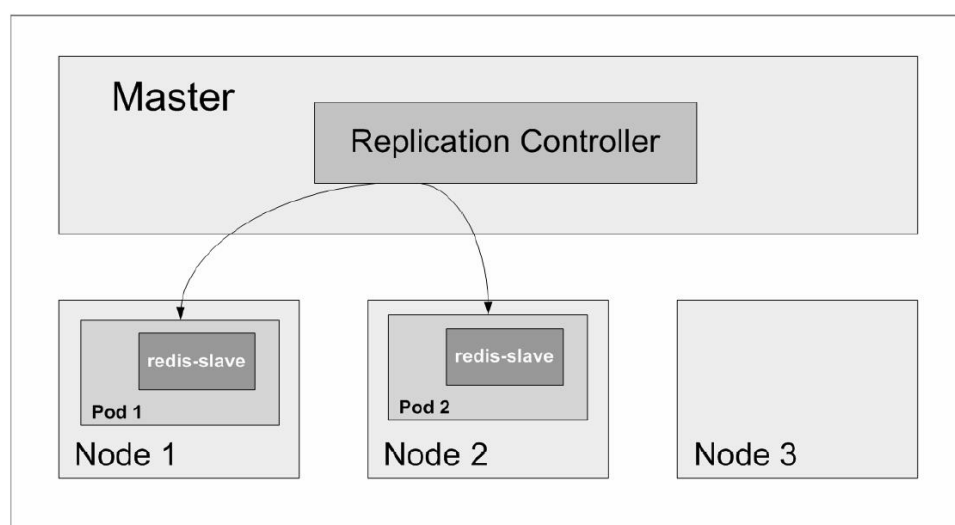


图1.11 在两个Node上创建redis-slave Pod

假设Node 2上的Pod 2意外终止，根据RC定义的replicas数量2，Kubernetes将会自动创建并启动一个新的Pod，以保证整个集群中始终有两个redis-slave Pod在运行。

如图1.12所示，系统可能选择Node 3或者Node 1来创建一个新的Pod。

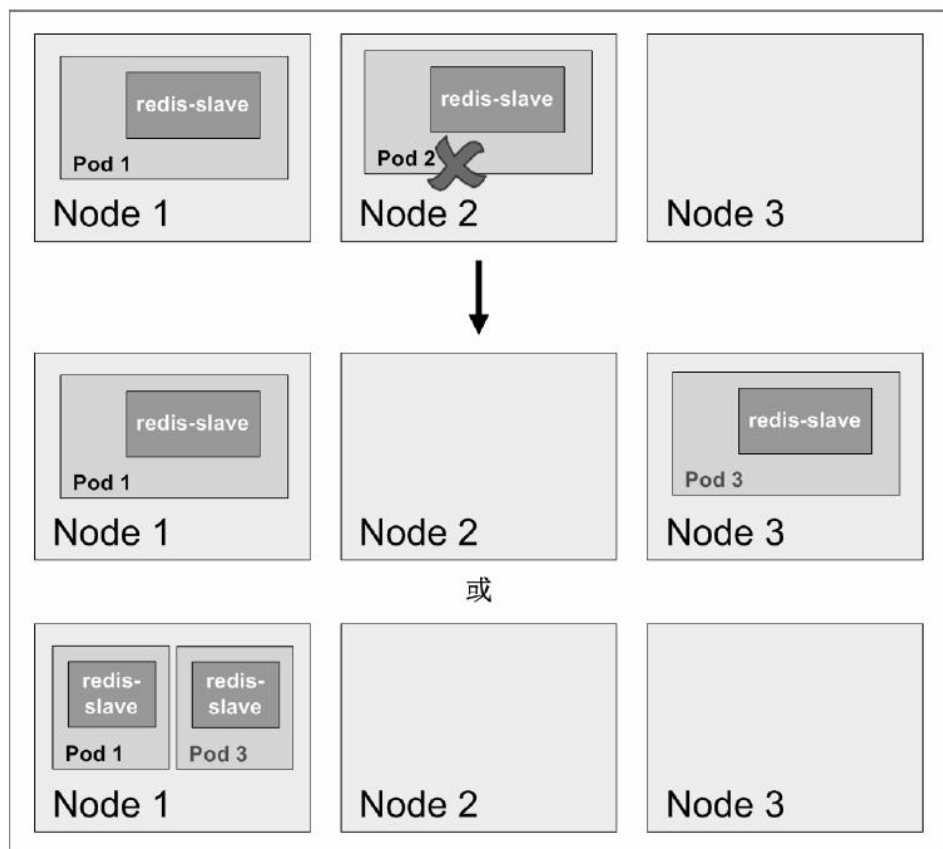


图1.12 根据RC定义创建新的Pod

此外，在运行时，我们可以通过修改RC的副本数量，来实现Pod的动态缩放（Scaling）功能，这可以通过执行kubect1 scale命令来一键完成：

```
$ kubect1 scale rc redis-slave --replicas=3  
scaled
```

Scaling的执行结果如图1.13所示。

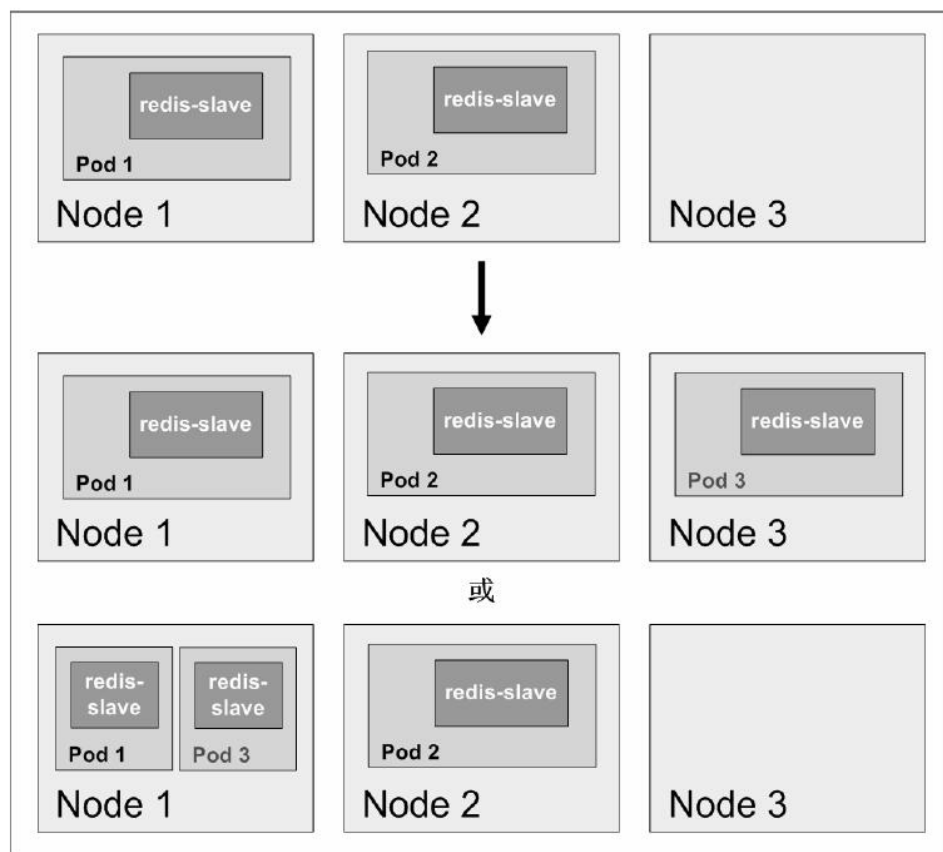


图1.13 Scaling的执行结果

需要注意的是，删除RC并不会影响通过该RC已创建好的Pod。为了删除所有Pod，可以设置replicas的值为0，然后更新该RC。另外，kubectl提供了stop和delete命令来一次性删除RC和RC控制的全部Pod。

当我们的应用升级时，通常会通过Build一个新的Docker镜像，并用新的镜像版本来替代旧的版本的方式达到目的。在系统升级的过程中，我们希望是平滑的方式，比如当前系统中10个对应的旧版本的Pod，最佳的方式是旧版本的Pod每次停止一个，同时创建一个新版本的Pod，在整个升级过程中，此消彼长，而运行中的Pod数量始终是10个，几分钟以后，当所有的Pod都已经是新版本的时候，升级过程完

成。通过RC的机制，Kubernetes很容易就实现了这种高级实用的特性，被称为“滚动升级”（Rolling Update），具体的操作方法详见第4章。

由于Replication Controller与Kubernetes代码中的模块Replication Controller同名，同时这个词也无法准确表达它的本意，所以在Kubernetes 1.2的时候，它就升级成了另外一个新的概念——Replica Set，官方解释为“下一代的RC”，它与RC当前存在的唯一区别是：Replica Sets支持基于集合的Label selector（Set-based selector），而RC只支持基于等式的Label Selector（equality-based selector），这使得Replica Set的功能更强，下面是等价于之前RC例子的Replica Set的定义（省去了Pod模板部分的内容）：

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: frontend
spec:
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    .....
```

kubectl命令行工具适用于RC的绝大部分命令都同样适用于Replica Set。此外，当前我们很少单独使用Replica Set，它主要被Deployment这个更高层的资源对象所使用，从而形成一整套Pod创建、删除、更新的编排机制。当我们使用Deployment时，无须关心它是如何创建和维护Replica Set的，这一切都是自动发生的。

Replica Set与Deployment这两个重要资源对象逐步替换了之前的RC的作用，是Kubernetes 1.3里Pod自动扩容（伸缩）这个告警功能实现的基础，也将继续在Kubernetes未来的版本中发挥重要的作用。

最后我们总结一下关于RC（Replica Set）的一些特性与作用。

- 在大多数情况下，我们通过定义一个RC实现Pod的创建过程及副本数量的自动控制。
- RC里包括完整的Pod定义模板。
- RC通过Label Selector机制实现对Pod副本的自动控制。
- 通过改变RC里的Pod副本数量，可以实现Pod的扩容或缩容功能。
- 通过改变RC里Pod模板中的镜像版本，可以实现Pod的滚动升级功能。

1.4.6 Deployment

Deployment是Kubernetes 1.2引入的新概念，引入的目的是为了更好地解决Pod的编排问题。为此，Deployment在内部使用了Replica Set来实现目的，无论从Deployment的作用与目的、它的YAM定义，还是从它的具体命令行操作来看，我们都可以把它看作RC的一次升级，两者的相似度超过90%。

Deployment相对于RC的一个最大升级是我们可以随时知道当前Pod“部署”的进度。实际上由于一个Pod的创建、调度、绑定节点及在目标Node上启动对应的容器这一完整过程需要一定的时间，所以我们期待系统启动N个Pod副本的目标状态，实际上是一个连续变化的“部署过程”导致的最终状态。

Deployment的典型使用场景有以下几个。

- 创建一个Deployment对象来生成对应的Replica Set并完成Pod副本的创建过程。
- 检查Deployment的状态来看部署动作是否完成（Pod副本的数量是否达到预期的值）。
- 更新Deployment以创建新的Pod（比如镜像升级）。
- 如果当前Deployment不稳定，则回滚到一个早先的Deployment版本。
- 挂起或者恢复一个Deployment。

Deployment的定义与Replica Set的定义很类似，除了API声明与Kind类型等有所区别：

<code>apiVersion: extensions/v1beta1</code>	<code>apiVersion: v1</code>
<code>kind: Deployment</code>	<code>kind: ReplicaSet</code>
<code>metadata:</code>	<code>metadata:</code>
<code> name: nginx-deployment</code>	<code> name: nginx-repset</code>

下面我们通过运行一些例子来一起直观地感受这个新概念。首先创建一个名为tomcat-deployment.yaml的Deployment描述文件，内容如下：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: app-demo
        tier: frontend
    spec:
      containers:
        - name: tomcat-demo
          image: tomcat
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 8080
```

运行下述命令创建Deployment:

```
# kubectl create -f tomcat-deployment.yaml
deployment "tomcat-deploy" created
```

运行下述命令查看Deployment的信息:

```
# kubectl get deployments
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
tomcat-deploy	1	1	1	1	4m

对上述输出中涉及的数量解释如下。

- **DESIRED**: Pod副本数量的期望值，即Deployment里定义的Replica。
- **CURRENT**: 当前Replica的值，实际上是Deployment所创建的Replica Set里的Replica值，这个值不断增加，直到达到**DESIRED**为止，表明整个部署过程完成。
- **UP-TO-DATE**: 最新版本的Pod的副本数量，用于指示在滚动升级的过程中，有多少个Pod副本已经成功升级。
- **AVAILABLE**: 当前集群中可用的Pod副本数量，即集群中当前存活的Pod数量。

运行下述命令查看对应的Replica Set，我们看到它的命名跟Deployment的名字有关系:

#kubectl get rs				
NAME	DESIRED	CURRENT	AGE	
tomcat-deploy-1640611518	1	1	1m	

运行下述命令查看创建的Pod，我们发现Pod的命名以Deployment对应的Replica Set的名字为前缀，这种命名很清晰地表明了一个Replica Set创建了哪些Pod，对于Pod滚动升级这种复杂的过程来说，很容易排查错误:

# kubectl get pods				
NAME	READY	STATUS	RESTARTS	AGE
tomcat-deploy-1640611518-zhrsc	1/1	Running	0	3m

运行kubectl describe deployments，可以清楚地看到Deployment控制的Pod的水平扩展过程，此命令的输出比较多，这里不再赘述。

1.4.7 Horizontal Pod Autoscaler (HPA)

在前两节提到过，通过手工执行`kubectl scale`命令，我们可以实现Pod扩容或缩容。如果仅仅到此为止，显然不符合谷歌对Kubernetes的定位目标——自动化、智能化。在谷歌看来，分布式系统要能够根据当前负载的变化情况自动触发水平扩展或缩容的行为，因为这一过程可能是频繁发生的、不可预料的，所以手动控制的方式是不现实的。

因此，Kubernetes的1.0版本实现后，这帮大牛们就已经在默默研究Pod智能扩容的特性了，并在Kubernetes1.1的版本中首次发布这一重量级新特性——Horizontal Pod Autoscaling。随后的1.2版本中HPA被升级为稳定版本（`apiVersion: autoscaling/v1`），但同时仍然保留旧版本（`apiVersion: extensions/v1beta1`），官方的计划是在1.3版本里先移除旧版本，并且会在1.4版本里彻底移除旧版本的支持。

Horizontal Pod Autoscaling简称HPA，意思是Pod横向自动扩容，与之前的RC、Deployment一样，也属于一种Kubernetes资源对象。通过追踪分析RC控制的所有目标Pod的负载变化情况，来确定是否需要针对性地调整目标Pod的副本数，这是HPA的实现原理。当前，HPA可以有以下两种方式作为Pod负载的度量指标。

- CPUUtilizationPercentage。
- 应用程序自定义的度量指标，比如服务在每秒内的相应的请求数（TPS或QPS）。

CPUUtilizationPercentage是一个算术平均值，即目标Pod所有副本自身的CPU利用率的平均值。一个Pod自身的CPU利用率是该Pod当前CPU的使用量除以它的Pod Request的值，比如我们定义一个Pod的Pod Request为0.4，而当前Pod的CPU使用量为0.2，则它的CPU使用率为50%，如此一来，我们就可以就算出来一个RC控制的所有Pod副本的CPU利用率的算术平均值了。如果某一时刻CPUUtilizationPercentage的值超过80%，则意味着当前的Pod副本数很可能不足以支撑接下来更多的请求，需要进行动态扩容，而当请求高峰时段过去后，Pod的CPU利用率又会降下来，此时对应的Pod副本数应该自动减少到一个合理的水平。

CPUUtilizationPercentage计算过程中使用到的Pod的CPU使用量通常是1分钟内的平均值，目前通过查询Heapster扩展组件来得到这个值，所以需要安装部署Heapster，这样一来便增加了系统的复杂度和实施HPA特性的复杂度，因此，未来的计划是Kubernetes自身实现一个基础性能数据采集模块，从而更好地支持HPA和其他需要用到基础性能数据的功能模块。此外，我们也看到，如果目标Pod没有定义Pod Request的值，则无法使用CPUUtilizationPercentage来实现Pod横向自动扩容的能力。除了使用CPUUtilizationPercentage，Kubernetes从1.2版本开始，尝试支持应用程序自定义的度量指标，目前仍然为实验特性，不建议在生产环境中使用。

下面是HPA定义的一个具体例子：

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
```

```
name: php-apache
namespace: default
spec:
  maxReplicas: 10
  minReplicas: 1
  scaleTargetRef:
    kind: Deployment
    name: php-apache
  targetCPUUtilizationPercentage: 90
```

根据上面的定义，我们可以知道这个**HPA**控制的目标对象为一个名叫 **php-apache** 的 **Deployment** 里的 **Pod** 副本，当这些 **Pod** 副本的 **CPUUtilizationPercentage** 的值超过90%时会触发自动动态扩容行为，扩容或缩容时必须满足的一个约束条件是**Pod**的副本数要介于1与10之间。

除了可以通过直接定义**yaml**文件并且调用**kubectl create**的命令来创建一个**HPA**资源对象的方式，我们还能通过下面的简单命令行直接创建等价的**HPA**对象：

```
# kubectl autoscale deployment php-apache --cpu-percent=90 --min=1 --max=10
```

第2章将会给出一个完整的**HPA**例子来说明其用法和功能。

1.4.8 Service（服务）

1.概述

Service也是Kubernetes里的最核心的资源对象之一，Kubernetes里的每个Service其实就是我们经常提起的微服务架构中的一个“微服务”，之前我们所说的Pod、RC等资源对象其实都是为这节所说的“服务”——Kubernetes Service做“嫁衣”的。图1.14显示了Pod、RC与Service的逻辑关系。

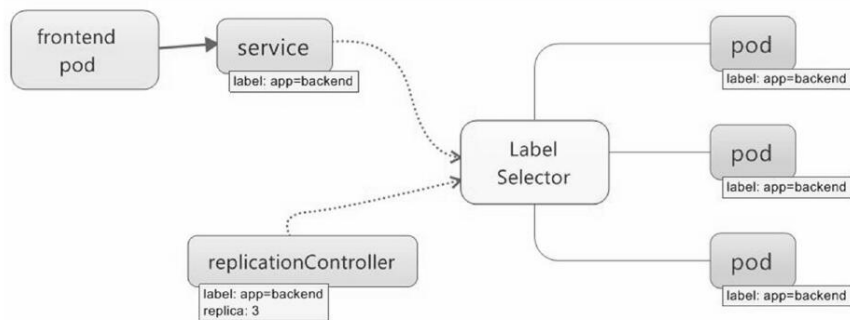


图1.14 Pod、RC与Service的关系

从图1.14中我们看到，Kubernetes的Service定义了一个服务的访问入口地址，前端的应用（Pod）通过这个入口地址访问其背后的一组由Pod副本组成的集群实例，Service与其后端Pod副本集群之间则是通过Label Selector来实现“无缝对接”的。而RC的作用实际上是保证Service的服务能力和服务质量始终处于预期的标准。

通过分析、识别并建模系统中的所有服务为微服务——**Kubernetes Service**，最终我们的系统由多个提供不同业务能力而又彼此独立的微服务单元所组成，服务之间通过**TCP/IP**进行通信，从而形成了我们强大而又灵活的弹性网格，拥有了强大的分布式能力、弹性扩展能力、容错能力，与此同时，我们的程序架构也变得简单和直观许多，如图1.15所示。

既然每个**Pod**都会被分配一个单独的**IP**地址，而且每个**Pod**都提供了一个独立的**Endpoint**（**Pod IP+ContainerPort**）以被客户端访问，现在多个**Pod**副本组成了一个集群来提供服务，那么客户端如何来访问它们呢？一般的做法是部署一个负载均衡器（软件或硬件），为这组**Pod**开启一个对外的服务端口如8000端口，并且将这些**Pod**的**Endpoint**列表加入8000端口的转发列表中，客户端就可以通过负载均衡器的对外**IP**地址+服务端口来访问此服务，而客户端的请求最后会被转发到哪个**Pod**，则由负载均衡器的算法所决定。

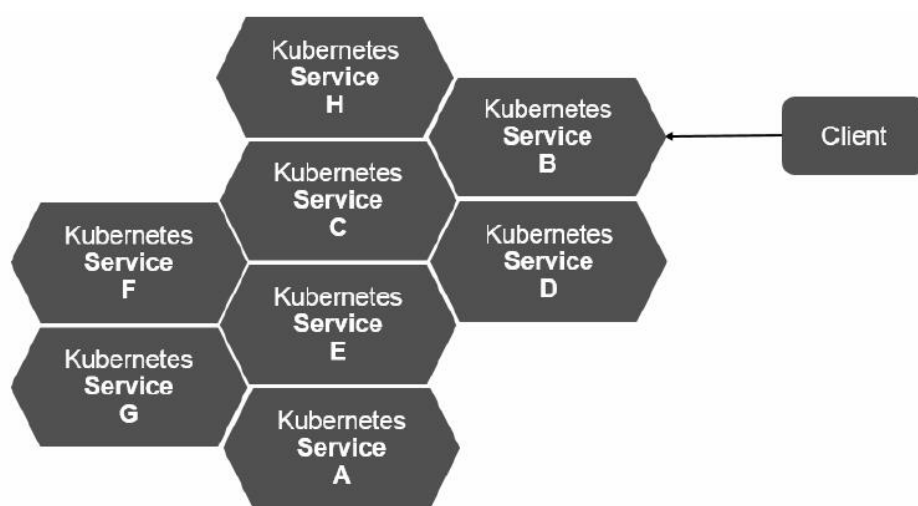


图1.15 Kubernetes所提供的微服务网格架构

Kubernetes也遵循了上述常规做法，运行在每个Node上的kube-proxy进程其实就是一个智能的软件负载均衡器，它负责把对Service的请求转发到后端的某个Pod实例上，并在内部实现服务的负载均衡与会话保持机制。但Kubernetes发明了一种很巧妙又影响深远的设计：Service不是共用一个负载均衡器的IP地址，而是每个Service分配了一个全局唯一的虚拟IP地址，这个虚拟IP被称为Cluster IP。这样一来，每个服务就变成了具备唯一IP地址的“通信节点”，服务调用就变成了最基础的TCP网络通信问题。

我们知道，Pod的Endpoint地址会随着Pod的销毁和重新创建而发生改变，因为新Pod的IP地址与之前旧Pod的不同。而Service一旦创建，Kubernetes就会自动为它分配一个可用的Cluster IP，而且在Service的整个生命周期内，它的Cluster IP不会发生改变。于是，服务发现这个棘手的问题在Kubernetes的架构里也得以轻松解决：只要用Service的Name与Service的Cluster IP地址做一个DNS域名映射即可完美解决问题。现在想想，这真是一个很棒的设计。

说了这么久，下面我们动手创建一个Service，来加深对它的理解。首先我们创建一个名为tomcat-service.yaml的定义文件，内容如下：

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
spec:
  ports:
```

```
- port: 8080
selector:
  tier: frontend
```

上述内容定义了一个名为“tomcat-service”的Service，它的服务端口为8080，拥有“tier=frontend”这个Label的所有Pod实例都属于它，运行下面的命令进行创建：

```
#kubectl create -f tomcat-server.yaml
service "tomcat-service" created
```

注意到我们之前在tomcat-deployment.yaml里定义的Tomcat的Pod刚好拥有这个标签，所以我们刚才创建的tomcat-service已经对应到了一个Pod实例，运行下面的命令可以查看tomcat-service的Endpoint列表，其中172.17.1.3是Pod的IP地址，端口8080是Container暴露的端口：

```
# kubectl get endpoints
```

NAME	ENDPOINTS	AGE
kubernetes	192.168.18.131:6443	15d
tomcat-service	172.17.1.3:8080	1m

你可能有疑问：“说好的Service的Cluster IP呢？怎么没有看到？”我们运行下面的命令即可看到tomcat-service被分配的Cluster IP及更多的信息：

```
# kubectl get svc tomcat-service -o yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: 2016-07-21T17:05:52Z
  name: tomcat-service
  namespace: default
  resourceVersion: "23964"
  selfLink:
    /api/v1/namespaces/default/services/tomcat-service
  uid: 61987d3c-4f65-11e6-a9d8-000c29ed42c1
spec:
  clusterIP: 169.169.65.227
  ports:
    - port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    tier: frontend
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

在spec.ports的定义中，targetPort属性用来确定提供该服务的容器所暴露（EXPOSE）的端口号，即具体业务进程在容器内的targetPort

上提供TCP/IP接入；而port属性则定义了Service的虚端口。前面我们定义Tomcat服务的时候，没有指定targetPort，则默认targetPort与port相同。

接下来，我们来看看Service的多端口问题。

很多服务都存在多个端口的问题，通常一个端口提供业务服务，另外一个端口提供管理服务，比如Mycat、Codis等常见中间件。Kubernetes Service支持多个Endpoint，在存在多个Endpoint的情况下，要求每个Endpoint定义一个名字来区分。下面是Tomcat多端口的Service定义样例：

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
spec:
  ports:
    - port: 8080
      name: service-port
    - port: 8005
      name: shutdown-port
  selector:
    tier: frontend
```

多端口为什么需要给每个端口命名呢？这就涉及Kubernetes的服务发现机制了，我们接下来进行讲解。

2.Kubernetes的服务发现机制

任何分布式系统都会涉及“服务发现”这个基础问题，大部分分布式系统通过提供特定的API接口来实现服务发现的功能，但这样做会导致平台的侵入性比较强，也增加了开发测试的困难。Kubernetes则采用了直观朴素的思路去解决这个棘手的问题。

首先，每个Kubernetes中的Service都有一个唯一的Cluster IP以及唯一的名字，而名字是由开发者自己定义的，部署的时候也没必要改变，所以完全可以固定在配置中。接下来的问题就是如何通过Service的名字找到对应的Cluster IP？

最早的时候Kubernetes采用了Linux环境变量的方式解决这个问题，即每个Service生成一些对应的Linux环境变量（ENV），并在每个Pod的容器在启动时，自动注入这些环境变量，以以下是tomcat-service产生的环境变量条目：

```
TOMCAT_SERVICE_SERVICE_HOST=169.169.41.218
TOMCAT_SERVICE_SERVICE_PORT_SERVICE_PORT=8080
TOMCAT_SERVICE_SERVICE_PORT_SHUTDOWN_PORT=8005
TOMCAT_SERVICE_SERVICE_PORT=8080
TOMCAT_SERVICE_PORT_8005_TCP_PORT=8005
TOMCAT_SERVICE_PORT=tcp://169.169.41.218:8080
TOMCAT_SERVICE_PORT_8080_TCP_ADDR=169.169.41.218
TOMCAT_SERVICE_PORT_8080_TCP=tcp://169.169.41.218:8080
TOMCAT_SERVICE_PORT_8080_TCP_PROTO=tcp
TOMCAT_SERVICE_PORT_8080_TCP_PORT=8080
TOMCAT_SERVICE_PORT_8005_TCP=tcp://169.169.41.218:8005
```

```
TOMCAT_SERVICE_PORT_8005_TCP_ADDR=169.169.41.218
```

```
TOMCAT_SERVICE_PORT_8005_TCP_PROTO=tcp
```

上述环境变量中，比较重要的是前3条环境变量，我们可以看到，每个Service的IP地址及端口都是有标准的命名规范的，遵循这个命名规范，就可以通过代码访问系统环境变量的方式得到所需的信息，实现服务调用。

考虑到环境变量的方式获取Service的IP与端口的方式仍然不太方便，不够直观，后来Kubernetes通过Add-On增值包的方式引入了DNS系统，把服务名作为DNS域名，这样一来，程序就可以直接使用服务名来建立通信连接了。目前Kubernetes上的大部分应用都已经采用了DNS这些新兴的服务发现机制，后面的章节中我们会讲述如何部署这套DNS系统。

3.外部系统访问Service的问题

为了更加深入地理解和掌握Kubernetes，我们需要弄明白Kubernetes里的“三种IP”这个关键问题，这三种IP分别如下。

- Node IP: Node节点的IP地址。
- Pod IP: Pod的IP地址。
- Cluster IP: Service的IP地址。

首先，Node IP是Kubernetes集群中每个节点的物理网卡的IP地址，这是一个真实存在的物理网络，所有属于这个网络的服务器之间都能通过这个网络直接通信，不管它们中是否有部分节点不属于这个Kubernetes集群。这也表明了Kubernetes集群之外的节点访问

Kubernetes集群之内的某个节点或者TCP/IP服务的时候，必须要通过Node IP进行通信。

其次，Pod IP是每个Pod的IP地址，它是Docker Engine根据docker0网桥的IP地址段进行分配的，通常是一个虚拟的二层网络，前面我们说过，Kubernetes要求位于不同Node上的Pod能够彼此直接通信，所以Kubernetes里一个Pod里的容器访问另外一个Pod里的容器，就是通过Pod IP所在的虚拟二层网络进行通信的，而真实的TCP/IP流量则是通过Node IP所在的物理网卡流出的。

最后，我们说说Service的Cluster IP，它也是一个虚拟的IP，但更像一个“伪造”的IP网络，原因有以下几点。

- Cluster IP仅仅作用于Kubernetes Service这个对象，并由Kubernetes管理和分配IP地址（来源于Cluster IP地址池）。
- Cluster IP无法被Ping，因为没有有一个“实体网络对象”来响应。
- Cluster IP只能结合Service Port组成一个具体的通信端口，单独的Cluster IP不具备TCP/IP通信的基础，并且它们属于Kubernetes集群这样一个封闭的空间，集群之外的节点如果要访问这个通信端口，则需要做一些额外的工作。
- 在Kubernetes集群之内，Node IP网、Pod IP网与Cluster IP网之间的通信，采用的是Kubernetes自己设计的一种编程方式的特殊的路由规则，与我们所熟知的IP路由有很大的不同。

根据上面的分析和总结，我们基本明白了：Service的Cluster IP属于Kubernetes集群内部的地址，无法在集群外部直接使用这个地址。那么矛盾来了：实际上我们开发的业务系统中肯定多少有一部分服务是要提供给Kubernetes集群外部的应用或者用户来使用的，典型的例

子就是Web端的服务模块，比如上面的tomcat-service，那么用户怎么访问它？

采用NodePort是解决上述问题的最直接、最有效、最常用的做法。具体做法如下，以tomcat-service为例，我们在Service的定义里做如下扩展即可（黑体字部分）：

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
spec:
  type: NodePort
  ports:
    - port: 8080
      nodePort: 31002
  selector:
    tier: frontend
```

其中，nodePort: 31002这个属性表明我们手动指定tomcat-service的NodePort为31002，否则Kubernetes会自动分配一个可用的端口。接下来，我们在浏览器里访问http://<nodePort IP>: 31002/，就可以看到Tomat的欢迎界面了，如图1.16所示。

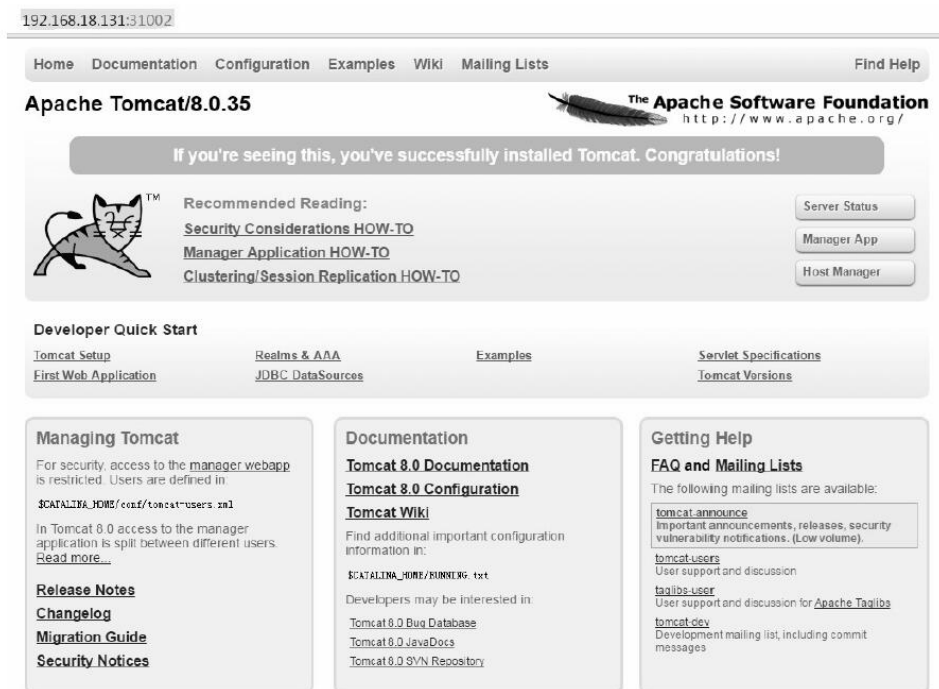


图1.16 通过NodePort访问Service

NodePort的实现方式是在Kubernetes集群里的每个Node上为需要外部访问的Service开启一个对应的TCP监听端口，外部系统只要用任意一个Node的IP地址+具体的NodePort端口号即可访问此服务，在任意Node上运行netstat命令，我们就可以看到有NodePort端口被监听：

```
# netstat -tlnp|grep 31002
```

tcp6	0	0	[::]:31002	[::]:*
------	---	---	------------	--------

```
LISTEN      1125/kube-proxy
```

但NodePort还没有完全解决外部访问Service的所有问题，比如负载均衡问题，假如我们的集群中有10个Node，则此时最好有一个负载均衡器，外部的请求只需访问此负载均衡器的IP地址，由负载均衡器负责转发流量到后面某个Node的NodePort上。如图1.17所示。

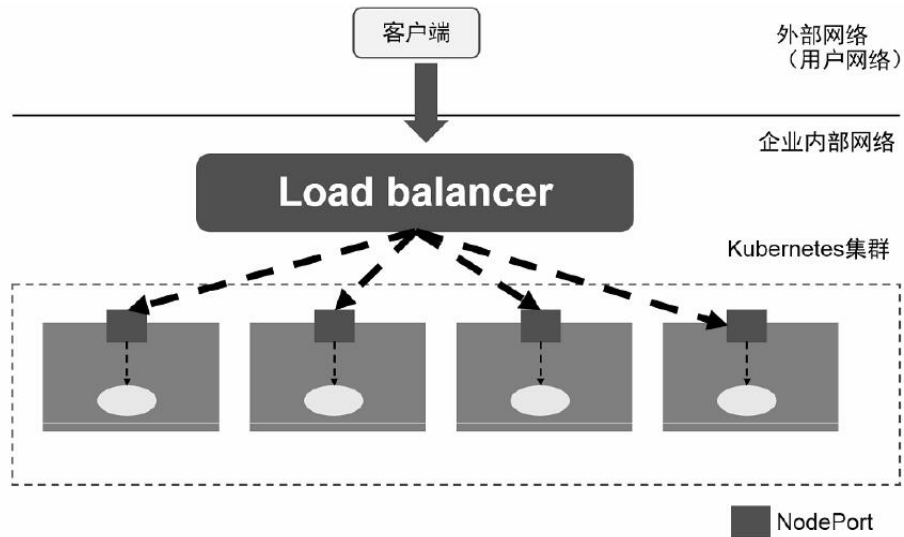


图1.17 NodePort与Load balancer

图1.17中的Load balancer组件独立于Kubernetes集群之外，通常是一个硬件的负载均衡器，或者是以软件方式实现的，例如HAProxy或者Nginx。对于每个Service，我们通常需要配置一个对应的Load balancer实例来转发流量到后端的Node上，这的确增加了工作量及出错的概率。于是Kubernetes提供了自动化的解决方案，如果我们的集群运行在谷歌的GCE公有云上，那么只要我们把Service的type=NodePort改为type=LoadBalancer，此时Kubernetes会自动创建一个对应的Load balancer实例并返回它的IP地址供外部客户端使用。其他公有云提供商只要实现了支持此特性的驱动，则也可以达到上述目的。此外，裸机上的类似机制（Bare Metal Service Load Balancers）也正在被开发。

1.4.9 Volume（存储卷）

Volume是Pod中能够被多个容器访问的共享目录。Kubernetes的Volume概念、用途和目的与Docker的Volume比较类似，但两者不能等价。首先，Kubernetes中的Volume定义在Pod上，然后被一个Pod里的多个容器挂载到具体的文件目录下；其次，Kubernetes中的Volume与Pod的生命周期相同，但与容器的生命周期不相关，当容器终止或者重启时，Volume中的数据也不会丢失。最后，Kubernetes支持多种类型的Volume，例如GlusterFS、Ceph等先进的分布式文件系统。

Volume的使用也比较简单，在大多数情况下，我们先在Pod上声明一个Volume，然后在容器里引用该Volume并Mount到容器里的某个目录上。举例来说，我们要给之前的Tomcat Pod增加一个名字为dataVol的Volume，并且Mount到容器的/mydata-data目录上，则只要对Pod的定义文件做如下修正即可（注意黑体字部分）：

```
template:
  metadata:
    labels:
      app: app-demo
      tier: frontend
  spec:
    volumes:
      - name: dataVol
        emptyDir: {}
```

```
containers:
- name: tomcat-demo
  image: tomcat
  volumeMounts:
    - mountPath: /mydata-data
      name: datavol
  imagePullPolicy: IfNotPresent
```

除了可以让一个Pod里的多个容器共享文件、让容器的数据写到宿主机的磁盘上或者写文件到网络存储中，Kubernetes的Volume还扩展出了一种非常有实用价值的功能，即容器配置文件集中化定义与管理，这是通过ConfigMap这个新的资源对象来实现的，后面我们会详细说明。

Kubernetes提供了非常丰富的Volume类型，下面逐一进行说明。

1.emptyDir

一个emptyDir Volume是在Pod分配到Node时创建的。从它的名称就可以看出，它的初始内容为空，并且无须指定宿主机上对应的目录文件，因为这是Kubernetes自动分配的一个目录，当Pod从Node上移除时，emptyDir中的数据也会被永久删除。emptyDir的一些用途如下。

- 临时空间，例如用于某些应用程序运行时所需的临时目录，且无须永久保留。
- 长时间任务的中间过程CheckPoint的临时保存目录。
- 一个容器需要从另一个容器中获取数据的目录（多容器共享目录）。

目前，用户无法控制emptyDir使用的介质种类。如果kubelet的配置是使用硬盘，那么所有emptyDir都将创建在该硬盘上。Pod在将来可以设置emptyDir是位于硬盘、固态硬盘上还是基于内存的tmpfs上，上面的例子便采用了emptyDir类的Volume。

2.hostPath

hostPath为在Pod上挂载宿主机上的文件或目录，它通常可以用于以下几方面。

- 容器应用程序生成的日志文件需要永久保存时，可以使用宿主机的高速文件系统进行存储。
- 需要访问宿主机上Docker引擎内部数据结构的容器应用时，可以通过定义hostPath为宿主机/var/lib/docker目录，使容器内部应用可以直接访问Docker的文件系统。

在使用这种类型的Volume时，需要注意以下几点。

- 在不同的Node上具有相同配置的Pod可能会因为宿主机上的目录和文件不同而导致对Volume上目录和文件的访问结果不一致。
- 如果使用了资源配额管理，则Kubernetes无法将hostPath在宿主机上使用的资源纳入管理。

在下面的例子中使用宿主机的/data目录定义了一个hostPath类型的Volume：

```
volumes:  
- name: "persistent-storage"
```

```
hostPath:
  path: "/data"
```

3.gcePersistentDisk

使用这种类型的Volume表示使用谷歌公有云提供的永久磁盘（Persistent Disk，PD）存放Volume的数据，它与EmptyDir不同，PD上的内容会被永久保存，当Pod被删除时，PD只是被卸载（Unmount），但不会被删除。需要注意的是，你需要先创建一个永久磁盘（PD），才能使用gcePersistentDisk。

使用gcePersistentDisk有以下一些限制条件。

- Node（运行kubelet的节点）需要是GCE虚拟机。
- 这些虚拟机需要与PD存在于相同的GCE项目和Zone中。

通过gcloud命令即可创建一个PD:

```
gcloud compute disks create --size=500GB --zone=us-central1-a my-data-disk
```

定义gcePersistentDisk类型的Volume的示例如下:

```
volumes:
- name: test-volume
  # This GCE PD must already exist.
  gcePersistentDisk:
```

```
pdName: my-data-disk
fsType: ext4
```

4.awsElasticBlockStore

与GCE类似，该类型的Volume使用亚马逊公有云提供的EBS Volume 存储数据，需要先创建一个EBSVolume才能使用awsElasticBlockStore。

使用awsElasticBlockStore的一些限制条件如下。

- Node（运行kubelet的节点）需要是AWS EC2实例。
- 这些AWS EC2实例需要与EBS volume存在于相同的region和availability-zone中。
- EBS只支持单个EC2实例mount一个volume。

通过aws ec2create-volume命令可以创建一个EBS volume:

```
aws ec2 create-volume --availability-zone eu-west-1a -
-size 10 --volume-type gp2
```

定义awsElasticBlockStore类型的Volume的示例如下:

```
volumes:
- name: test-volume
  # This AWS EBS volume must already exist.
  awsElasticBlockStore:
```

```
volumeID: aws://<availability-zone>/<volume-id>
fsType: ext4
```

5.NFS

使用NFS网络文件系统提供的共享目录存储数据时，我们需要在系统中部署一个NFS Server。定义NFS类型的Volume的示例如下：

```
volumes:
  - name: nfs
    nfs:
      # 改为你的NFS服务器地址
      server: nfs-server.localhost
      path: "/"
```

6.其他类型的Volume

- **iscsi**: 使用iSCSI存储设备上的目录挂载到Pod中。
- **flocker**: 使用Flocker来管理存储卷。
- **glusterfs**: 使用开源GlusterFS网络文件系统的目录挂载到Pod中。
- **rbd**: 使用Linux块设备共享存储（Rados Block Device）挂载到Pod中。
- **gitRepo**: 通过挂载一个空目录，并从GIT库clone一个git repository以供Pod使用。
- **secret**: 一个secret volume用于为Pod提供加密的信息，你可以将定义在Kubernetes中的secret直接挂载为文件让Pod访问。secret

volume是通过**tmfs**（内存文件系统）实现的，所以这种类型的**volume**总是不会持久化的。

1.4.10 Persistent Volume

之前我们提到的**Volume**是定义在**Pod**上的，属于“计算资源”的一部分，而实际上，“网络存储”是相对独立于“计算资源”而存在的一种实体资源。比如在使用虚机的情况下，我们通常会先定义一个网络存储，然后从中划出一个“网盘”并挂接到虚机上。**Persistent Volume**（简称**PV**）和与之相关联的**Persistent Volume Claim**（简称**PVC**）也起到了类似的作用。

PV可以理解成**Kubernetes**集群中的某个网络存储中对应的一块存储，它与**Volume**很类似，但有以下区别。

- **PV**只能是网络存储，不属于任何**Node**，但可以在每个**Node**上访问。
- **PV**并不是定义在**Pod**上的，而是独立于**Pod**之外定义。
- **PV**目前只有几种类型：**GCE Persistent Disks**、**NFS**、**RBD**、**iSCSI**、**AWS ElasticBlockStore**、**GlusterFS**等。

下面给出了**NFS**类型**PV**的一个**yaml**定义文件，声明了需要**5Gi**的存储空间：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
```



```
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  nfs:
    path: /somepath
    server: 172.17.0.2
```

比较重要的是PV的accessModes属性，目前有以下类型。

- ReadWriteOnce: 读写权限、并且只能被单个Node挂载。
- ReadOnlyMany: 只读权限、允许被多个Node挂载。
- ReadWriteMany: 读写权限、允许被多个Node挂载。

如果某个Pod想申请某种条件的PV，则首先需要定义一个PersistentVolumeClaim（PVC）对象：

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
```

然后，在Pod的Volume定义中引用上述PVC即可：

```
volumes:
  - name: mypd
    persistentVolumeClaim:
      claimName: myclaim
```

最后，我们说说PV的状态，PV是有状态的对象，它有以下几种状态。

- Available: 空闲状态。
- Bound: 已经绑定到某个PVC上。
- Released: 对应的PVC已经删除，但资源还没有被集群收回。
- Failed: PV自动回收失败。

1.4.11 Namespace（命名空间）

Namespace（命名空间）是Kubernetes系统中的另一个非常重要的概念，Namespace在很多情况下用于实现多租户的资源隔离。Namespace通过将集群内部的资源对象“分配”到不同的Namespace中，形成逻辑上分组的不同项目、小组或用户组，便于不同的分组在共享使用整个集群的资源的同时还能被分别管理。

Kubernetes 集群在启动后，会创建一个名为“default”的Namespace，通过kubectl可以查看到：

```
$ kubectl get namespaces
```

NAME	LABELS	STATUS
default	<none>	Active

接下来，如果不特别指明Namespace，则用户创建的Pod、RC、Service都将被系统创建到这个默认的名为default的Namespace中。

Namespace的定义很简单。如下所示的yaml定义了名为development的Namespace。

```
apiVersion: v1
kind: Namespace
metadata:
  name: development
```

一旦创建了Namespace，我们在创建资源对象时就可以指定这个资源对象属于哪个Namespace。比如在下面的例子中，我们定义了一个名为busybox的Pod，放入development这个Namespace里：

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: development
spec:
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    name: busybox
```

此时，使用kubectl get命令查看将无法显示：

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

这是因为如果不加参数，则kubectl get命令将仅显示属于“default”命名空间的资源对象。

可以在kubectl命令中加入--namespace参数来查看某个命名空间中的对象：

# kubectl get pods --namespace=development				
NAME	READY	STATUS	RESTARTS	AGE
busybox	1/1	Running	0	1m

当我们给每个租户创建一个Namespace来实现多租户的资源隔离时，还能结合Kubernetes的资源配额管理，限定不同租户能占用的资源，例如CPU使用量、内存使用量等。关于资源配额管理的问题，在后面的章节中会详细介绍。

1.4.12 Annotation（注解）

Annotation与Label类似，也使用key/value键值对的形式进行定义。不同的是Label具有严格的命名规则，它定义的是Kubernetes对象的元数据（Metadata），并且用于Label Selector。而Annotation则是用户任意定义的“附加”信息，以便于外部工具进行查找，很多时候，Kubernetes的模块自身会通过Annotation的方式标记资源对象的一些特殊信息。

通常来说，用Annotation来记录的信息如下。

- build信息、release信息、Docker镜像信息等，例如时间戳、release id号、PR号、镜像hash值、docker registry地址等。
- 日志库、监控库、分析库等资源库的地址信息。
- 程序调试工具信息，例如工具名称、版本号等。
- 团队的联系信息，例如电话号码、负责人名称、网址等。

1.4.13 小结

上述这些组件是Kubernetes系统的核心组件，它们共同构成了Kubernetes系统的框架和计算模型。通过对它们进行灵活组合，用户就可以快速、方便地对容器集群进行配置、创建和管理。除了本章所介绍的核心组件，在Kubernetes系统中还有许多辅助配置的资源对象，例如LimitRange、ResourceQuota。另外，一些系统内部使用的对象Binding、Event等请参考Kubernetes的API文档。

在第2章中，我们将开始深入实践并全面掌握Kubernetes的各种使用技巧。

第2章 Kubernetes实践指南

本章将从Kubernetes的系统安装开始，逐步介绍Kubernetes的服务相关配置、命令行工具kubectl的使用详解，然后通过大量案例实践对Kubernetes最核心的容器和微服务架构的概念和用法进行详细说明。

2.1 Kubernetes安装与配置

2.1.1 安装Kubernetes

Kubernetes系统由一组可执行程序组成，用户可以通过GitHub上的Kubernetes项目页下载编译好的二进制包，或者下载源代码并编译后进行安装。

安装Kubernetes对软件和硬件的系统要求如表2.1所示。

表2.1 安装Kubernetes对软件和硬件的系统要求

软 硬 件	最 低 配 置	推 荐 配 置
CPU 和内存	Master: 至少 1 core 和 2GB 内存 Node: 至少 1 core 和 2GB 内存	Master: 2 core 和 4GB 内存 Node: 由于要运行 Docker, 所以应根据需要运行的容器数量进行调整
Linux 操作 系统	基于 x86_64 架构的各种 Linux 发行版本, 包括 Red Hat Linux、CentOS、Fedora、Ubuntu 等, Kernel 版本要求在 3.10 及以上。 也可以在谷歌的 GCE(Google Compute Engine)或者 Amazon 的 AWS (Amazon Web Service) 云平台上进行安装	Red Hat Linux 7 CentOS 7

续表

软 硬 件	最 低 配 置	推 荐 配 置
Docker	1.9 版本及以上 下载和安装说明见 https://www.docker.com	1.12 版本
etcd	2.0 版本及以上 下载和安装说明见 https://github.com/coreos/etcd/releases	3.0 版本

最简单的安装方法是使用 `yum install kubernetes` 命令完成 Kubernetes 集群的安装，但仍需修改各组件的启动参数，才能完成 Kubernetes 集群的配置。

本章以二进制文件和手工配置启动参数的形式进行安装，对每个组件的配置进行详细说明。

从Kubernetes官网下载编译好的二进制包，如图2.1所示，下载地址为 <https://github.com/kubernetes/kubernetes/releases>。本书基于Kubernetes 1.3版本进行说明。



图2.1 GitHub上Kubernetes的下载页面

在压缩包kubernetes.tar.gz内包含了Kubernetes的服务程序文件、文档和示例。

解压缩后，server子目录中的kubernetes-server-linux-amd64.tar.gz文件包含了Kubernetes需要运行的全部服务程序文件。服务程序文件列表如表2.2所示。

表2.2 服务程序文件列表

文 件 名	说 明
hyperkube	总控程序，用于运行其他 Kubernetes 程序
kube-apiserver	apiserver 主程序
kube-apiserver.docker_tag	apiserver docker 镜像的 tag
kube-apiserver.tar	apiserver docker 镜像文件
kube-controller-manager	controller-manager 主程序
kube-controller-manager.docker_tag	controller-manager docker 镜像的 tag
kube-controller-manager.tar	controller-manager docker 镜像文件
kubecttl	客户端命令行工具

续表

文 件 名	说 明
kubelet	kubelet 主程序
kube-proxy	proxy 主程序
kube-scheduler	scheduler 主程序
kube-scheduler.docker_tag	scheduler docker 镜像的 tag
kube-scheduler.tar	scheduler docker 镜像文件

Kubernetes Master 节点安装部署 etcd 、 kube-apiserver 、 kube-controller-manager 、 kube-scheduler服务进程。我们使用kubecttl作为客户端与Master进行交互操作，在工作Node上仅需部署kubelet和kube-proxy服务进程。Kubernetes还提供了“all-in-one”的hyperkube程序来完成对以上服务程序的启动。

2.1.2 配置和启动Kubernetes服务

Kubernetes的服务都可以通过直接运行二进制文件加上启动参数完成。为了便于管理，常见的做法是将Kubernetes服务程序配置为Linux的系统开机自动启动的服务。

本节以CentOS Linux 7为例，使用Systemd系统完成Kubernetes服务的配置。其他Linux发行版的服务配置请参考相关的系统管理手册。

需要注意的是，CentOS Linux 7默认启动了firewalld——防火墙服务，而Kubernetes的Master与工作Node之间会有大量的网络通信，安全的做法是在防火墙上配置各组件需要相互通信的端口号，具体要配置的端口号详见2.1.6节中各服务监听的端口号说明。在一个安全的内部网络环境中可以关闭防火墙服务：

```
# systemctl disable firewalld
# systemctl stop firewalld
```

将Kubernetes的可执行文件复制到/usr/bin（如果复制到其他目录，则将systemd服务文件中的文件路径修改正确即可），然后对服务进行配置。

在下面的服务启动参数说明中主要介绍最重要的启动参数，每个服务的启动参数还有很多，详见2.1.6节的完整说明。有兴趣的读者可以尝试修改它们，以观察服务运行的不同效果。

1.Master上的etcd、kube-apiserver、kube-controller-manager、kube-scheduler服务

1) etcd服务

etcd服务作为Kubernetes集群的主数据库，在安装Kubernetes各服务之前需要首先安装和启动。

从GitHub官网下载etcd发布的二进制文件，将etcd和etcdctl文件复制到/usr/bin目录。

设置systemd服务文件/usr/lib/systemd/system/etcd.service:

```
[Unit]
Description=Etcd Server
After=network.target

[Service]
Type=simple
WorkingDirectory=/var/lib/etcd/
EnvironmentFile=-/etc/etcd/etcd.conf
ExecStart=/usr/bin/etcd

[Install]
WantedBy=multi-user.target
```

其中WorkingDirectory (/var/lib/etcd/) 表示etcd数据保存的目录，需要在启动etcd服务之前进行创建。

配置文件/etc/etcd/etcd.conf通常不需要特别的参数设置（详细的参数配置内容参见官方文档），etcd默认将监听在http://127.0.0.1: 2379地址供客户端连接。

配置完成后，通过systemctl start命令启动etcd服务。同时，使用systemctl enable命令将服务加入开机启动列表中：

```
# systemctl daemon-reload
# systemctl enable etcd.service
# systemctl start etcd.service
```

通过执行etcdctl cluster-health，可以验证etcd是否正确启动：

```
# etcdctl cluster-health
member ce2a822cea30bfca is healthy: got healthy result
from http://127.0.0.1:2379
cluster is healthy
```

2) kube-apiserver服务

将kube-apiserver的可执行文件复制到/usr/bin目录。

编辑 systemd 服务文件 /usr/lib/systemd/system/kube-apiserver.service，内容如下：

```
[Unit]
Description=Kubernetes API Server
```

Documentation=<https://github.com/GoogleCloudPlatform/kubernetes>

After=etcd.service

Wants=etcd.service

[Service]

EnvironmentFile=/etc/kubernetes/apiserver

ExecStart=/usr/bin/kube-apiserver \$KUBE_API_ARGS

Restart=on-failure

Type=notify

LimitNOFILE=65536

[Install]

WantedBy=multi-user.target

配置文件/etc/kubernetes/apiserver的内容包括了kube-apiserver的全部启动参数，主要的配置参数在变量KUBE_API_ARGS中指定。

```
# cat /etc/kubernetes/apiserver

KUBE_API_ARGS="--etcd_servers=http://127.0.0.1:2379 --
insecure-bind-address=0.0.0.0      --insecure-port=8080      --
service-cluster-ip-range=169.169.0.0/16      --service-node-
port-range=1-65535      --
admission_control=NamespaceLifecycle,LimitRanger,SecurityCo
ntextDeny,ServiceAccount,ResourceQuota      --logtostderr=false
--log-dir=/var/log/kubernetes --v=2"
```

对启动参数的说明如下。

- `--etcd_servers`: 指定etcd服务的URL。
- `--insecure-bind-address`: apiserver绑定主机的非安全IP地址，设置0.0.0.0表示绑定所有IP地址。
- `--insecure-port`: apiserver绑定主机的非安全端口号，默认为8080。
- `--service-cluster-ip-range`: Kubernetes集群中Service的虚拟IP地址段范围，以CIDR格式表示，例如169.169.0.0/16，该IP范围不能与物理机的真实IP段有重合。
- `--service-node-port-range`: Kubernetes集群中Service可映射的物理机端口号范围，默认为30000~32767。
- `--admission_control`: Kubernetes集群的准入控制设置，各控制模块以插件的形式依次生效。
- `--logtostderr`: 设置为false表示将日志写入文件，不写入stderr。
- `--log-dir`: 日志目录。
- `--v`: 日志级别。

3) kube-controller-manager服务

kube-controller-manager服务依赖于kube-apiserver服务。

```
# cat /usr/lib/systemd/system/kube-controller-  
manager.service  
  
[Unit]  
  
Description=Kubernetes Controller Manager  
  
  
Documentation=https://github.com/GoogleCloudPlatform/kubern
```



```
etes
    After=kube-apiserver.service
    Requires=kube-apiserver.service

    [Service]
    EnvironmentFile=/etc/kubernetes/controller-manager
    ExecStart=/usr/bin/kube-controller-manager
    $KUBE_CONTROLLER_MANAGER_ARGS
    Restart=on-failure
    LimitNOFILE=65536

    [Install]
    WantedBy=multi-user.target
```

配置文件 `/etc/kubernetes/controller-manager` 的内容包括了 `kube-controller-manager` 的全部启动参数，主要的配置参数在变量 `KUBE_CONTROLLER_MANAGER_ARGS` 中指定。

```
# cat /etc/kubernetes/controller-manager
                                KUBE_CONTROLLER_MANAGER_ARGS="--
master=http://192.168.18.3:8080 --logtostderr=false --log-
dir=/var/log/kubernetes --v=2"
```

对启动参数的说明如下。

- `--master`: 指定 `apiserver` 的 URL 地址。
- `--logtostderr`: 设置为 `false` 表示将日志写入文件，不写入 `stderr`。

- --log-dir: 日志目录。
- --v: 日志级别。

4) kube-scheduler服务

kube-scheduler服务也依赖于kube-apiserver服务。

```
# cat /usr/lib/systemd/system/kube-controller-
manager.service

[Unit]
Description=Kubernetes Controller Manager

Documentation=https://github.com/GoogleCloudPlatform/kubern
etes

After=kube-apiserver.service
Requires=kube-apiserver.service

[Service]
EnvironmentFile=/etc/kubernetes/scheduler
ExecStart=/usr/bin/kube-scheduler $KUBE_SCHEDULER_ARGS
Restart=on-failure
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
```

配置文件/etc/kubernetes/scheduler的内容包括了kube-scheduler的全部启动参数，主要的配置参数在变量KUBE_SCHEDULER_ARGS中指

定。

```
# cat /etc/kubernetes/scheduler

KUBE_SCHEDULER_ARGS="--master=http://192.168.18.3:8080
--logtostderr=false --log-dir=/var/log/kubernetes --v=2"
```

对启动参数的说明如下。

- `--master`: 指定apiserver的URL地址。
- `--logtostderr`: 设置为false表示将日志写入文件，不写入stderr。
- `--log-dir`: 日志目录。
- `--v`: 日志级别。

配置完成后，执行systemctl start命令按顺序启动这3个服务。同时，使用systemctl enable命令将服务加入开机启动列表中。

```
# systemctl daemon-reload
# systemctl enable kube-apiserver.service
# systemctl start kube-apiserver.service
# systemctl enable kube-controller-manager
# systemctl start kube-controller-manager
# systemctl enable kube-scheduler
# systemctl start kube-scheduler
```

通过 `systemctl status<service_name>` 来验证服务的启动状态，“running”表示启动成功。

到此，Master上所需的服务就全部启动完成了。

2.Node上的kubelet、kube-proxy服务

在工作Node节点上需要预先安装好Docker Daemon并且正常启动。Docker的安装详见<http://www.docker.com>的说明。

1) kubelet服务

kubelet服务依赖于Docker服务。

```
# cat /usr/lib/systemd/system/kubelet.service

[Unit]
Description=Kubernetes Kubelet Server

Documentation=https://github.com/GoogleCloudPlatform/kubernet
etes

After=docker.service
Requires=docker.service

[Service]
WorkingDirectory=/var/lib/kubelet
EnvironmentFile=/etc/kubernetes/kubelet
ExecStart=/usr/bin/kubelet $KUBELET_ARGS
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

其中WorkingDirectory表示kubelet保存数据的目录，需要在启动kubelet服务之前进行创建。

配置文件/etc/kubernetes/kubelet的内容包括了kubelet的全部启动参数，主要的配置参数在变量KUBELET_ARGS中指定。

```
# cat /etc/kubernetes/kubelet

KUBELET_ARGS="--api-servers=http://192.168.18.3:8080 -
-hostname-override=192.168.18.3 --logtostderr=false --log-
dir=/var/log/kubernetes --v=2"
```

对启动参数的说明如下。

- --api-servers: 指定apiserver的URL地址，可以指定多个。
- --hostname-override: 设置本Node的名称。
- --logtostderr: 设置为false表示将日志写入文件，不写入stderr。
- --log-dir: 日志目录。
- --v: 日志级别。

2) kube-proxy服务

kube-proxy服务依赖于network服务。

[Unit]

Description=Kubernetes Kube-Proxy Server

Documentation=<https://github.com/GoogleCloudPlatform/kubernetes>

```
After=network.target
Requires=network.service

[Service]
EnvironmentFile=/etc/kubernetes/proxy
ExecStart=/usr/bin/kube-proxy $KUBE_PROXY_ARGS
Restart=on-failure
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
```

配置文件/etc/kubernetes/proxy的内容包括了kube-proxy的全部启动参数，主要的配置参数在变量KUBE_PROXY_ARGS中指定。

```
# cat /etc/kubernetes/proxy

KUBE_PROXY_ARGS="--master=http://192.168.18.3:8080 --logtostderr=false --log-dir=/var/log/kubernetes --v=2"
```

对启动参数的说明如下。

- --master: 指定apiserver的URL地址。
- --logtostderr: 设置为false表示将日志写入文件，不写入stderr。
- --log-dir: 日志目录。
- --v: 日志级别。

配置完成后，通过systemctl启动kubelet和kube-proxy服务：

```
# systemctl daemon-reload
# systemctl enable kubelet.service
# systemctl start kubelet.service
# systemctl enable kube-proxy
# systemctl start kube-proxy
```

kubelet默认采用向Master自动注册本Node的机制，在Master上查看各Node的状态，状态为Ready表示Node已经成功注册并且状态为可用。

```
# kubectl get nodes
```

NAME	STATUS	AGE
192.168.18.3	Ready	1m

等所有Node的状态都为Ready之后，一个Kubernetes集群就启动完成了。接下来就可以创建Pod、RC、Service等资源对象来部署Docker容器应用了。

2.1.3 Kubernetes集群的安全设置

1. 基于CA签名的双向数字证书认证方式

在一个安全的内网环境中，Kubernetes的各个组件与Master之间可以通过apiserver的非安全端口`http://apiserver: 8080`进行访问。但如果apiserver需要对外提供服务，或者集群中的某些容器也需要访问apiserver以获取集群中的某些信息，则更安全的做法是启用HTTPS安全机制。Kubernetes提供了基于CA签名的双向数字证书认证方式和简单的基于HTTP BASE或TOKEN的认证方式，其中CA证书方式的安全性最高。本节先介绍以CA证书的方式配置Kubernetes集群，要求Master上的kube-apiserver、kube-controller-manager、kube-scheduler进程及各Node上的kubelet、kube-proxy进程进行CA签名双向数字证书安全设置。

基于CA签名的双向数字证书的生成过程如下。

(1) 为kube-apiserver生成一个数字证书，并用CA证书进行签名。

(2) 为kube-apiserver进程配置证书相关的启动参数，包括CA证书（用于验证客户端证书的签名真伪）、自己的经过CA签名后的证书及私钥。

(3) 为每个访问Kubernetes API Server的客户端（如kube-controller-manager、kube-scheduler、kubelet、kube-proxy及调用API

Server的客户端程序kubectl等）进程生成自己的数字证书，也都用CA证书进行签名，在相关程序的启动参数里增加CA证书、自己的证书等相关参数。

1) 设置kube-apiserver的CA证书相关的文件和启动参数

使用OpenSSL工具在Master服务器上创建CA证书和私钥相关的文件：

```
# openssl genrsa -out ca.key 2048
# openssl req -x509 -new -nodes -key ca.key -subj
"/CN=yourcompany.com" -days 5000 -out ca.crt
# openssl genrsa -out server.key 2048
```

注意：生成ca.crt时，-subj参数中“/CN”的值通常为域名。

准备master_ssl.cnf文件，该文件用于x509v3版本的证书。在该文件中主要需要设置Master服务器的hostname（k8s-master）、IP地址（192.168.18.3），以及Kubernetes Master Service的虚拟服务名称（kubernetes.default等）和该虚拟服务的ClusterIP地址（169.169.0.1）。

master_ssl.cnf文件的示例如下：

```
[req]
req_extensions = v3_req
distinguished_name = req_distinguished_name
[req_distinguished_name]
```

```
[ v3_req ]
basicConstraints = CA:FALSE
        keyUsage = nonRepudiation, digitalSignature,
keyEncipherment
subjectAltName = @alt_names
[alt_names]
DNS.1 = kubernetes
DNS.2 = kubernetes.default
DNS.3 = kubernetes.default.svc
DNS.4 = kubernetes.default.svc.cluster.local
DNS.5 = k8s-master
IP.1 = 169.169.0.1
IP.2 = 192.168.18.3
```

基于master_ssl.cnf创建server.csr和server.crt文件。在生成server.csr时，-subj参数中“/CN”指定的名字需为Master所在的主机名。

```
# openssl req -new -key server.key -subj "/CN=k8s-
master" -config master_ssl.cnf -out server.csr
# openssl x509 -req -in server.csr -CA ca.crt -CAkey
ca.key -CAcreateserial -days 5000 -extensions v3_req -
extfile master_ssl.cnf -out server.crt
```

全部执行完后会生成6个文件：ca.crt、ca.key、ca.srl、server.crt、server.csr、server.key。

将这些文件复制到一个目录中（例如/var/run/kubernetes/），然后设置kube-apiserver的三个启动参数“--client-ca-file”“--tls-cert-file”和“--tls-private-key-file”，分别代表CA根证书文件、服务端证书文件和服务端私钥文件：

```
--client-ca-file=/var/run/kubernetes/ca.crt
--tls-private-key-file=/var/run/kubernetes/server.key
--tls-cert-file=/var/run/kubernetes/server.crt
```

同时，可以关掉非安全端口8080，设置安全端口为443（默认为6443）：

```
--insecure-port=0
--secure-port=443
```

最后重启kube-apiserver服务。

2) 设置kube-controller-manager的客户端证书、私钥和启动参数

```
$ openssl genrsa -out cs_client.key 2048
$ openssl req -new -key cs_client.key -subj "/CN=k8s-
node-1" -out cs_client.csr
$ openssl x509 -req -in cs_client.csr -CA ca.crt -
CAkey ca.key -CAcreateserial -out cs_client.crt -days 5000
```

其中，在生成cs_client.crt时，-CA参数和-CAkey参数使用的是apiserver的ca.crt和ca.key文件。然后将这些文件复制到一个目录中（例如/var/run/kubernetes/）。

接下来创建 `/etc/kubernetes/kubeconfig` 文件（`kube-controller-manager`与`kube-scheduler`共用），配置客户端证书等相关参数，内容如下：

```
apiVersion: v1
kind: Config
users:
- name: controllermanager
  user:
    client-certificate:
      /var/run/kubernetes/cs_client.crt
    client-key: /var/run/kubernetes/cs_client.key
clusters:
- name: local
  cluster:
    certificate-authority: /var/run/kubernetes/ca.crt
contexts:
- context:
    cluster: local
    user: controllermanager
    name: my-context
current-context: my-context
```

然后，设置`kube-controller-manager`服务的启动参数，注意，`--master`的地址为HTTPS安全服务地址，不使用非安全地址`http://192.168.18.3: 8080`。

```
--master=https://192.168.18.3:443
--
service_account_private_key_file=/var/run/kubernetes/server
.key
--root-ca-file=/var/run/kubernetes/ca.crt
--kubeconfig=/etc/kubernetes/kubeconfig
```

重启kube-controller-manager服务。

3) 设置kube-scheduler启动参数

kube-scheduler复用上一步kube-controller-manager创建的客户端证书，配置启动参数：

```
--master=https://192.168.18.3:443
--kubeconfig=/etc/kubernetes/kubeconfig
```

重启kube-scheduler服务。

4) 设置每台Node上kubelet的客户端证书、私钥和启动参数

首先复制kube-apiserver的ca.crt和ca.key文件到Node上，在生成kubelet_client.crt时-CA参数和-CAkey参数使用的是apiserver的ca.crt和ca.key文件。在生成kubelet_client.csr时-subj参数中的“/CN”设置为本Node的IP地址。

```
$ openssl genrsa -out kubelet_client.key 2048
$ openssl req -new -key kubelet_client.key -subj
```

```
"/CN=192.168.18.4" -out kubelet_client.csr
$ openssl x509 -req -in kubelet_client.csr -CA ca.crt
-CAkey ca.key -CAcreateserial -out kubelet_client.crt -days
5000
```

将这些文件复制到一个目录中（例如/var/run/kubernetes/）。

接下来创建/etc/kubernetes/kubeconfig文件（kubelet和kube-proxy进程共用），配置客户端证书等相关参数，内容如下：

```
apiVersion: v1
kind: Config
users:
- name: kubelet
  user:
    client-certificate:
      /etc/kubernetes/ssl_keys/kubelet_client.crt
    client-key:
      /etc/kubernetes/ssl_keys/kubelet_client.key
clusters:
- name: local
  cluster:
    certificate-authority:
      /etc/kubernetes/ssl_keys/ca.crt
contexts:
- context:
    cluster: local
```

```
user: kubelet
name: my-context
current-context: my-context
```

然后，设置kubelet服务的启动参数：

```
--api_servers=https://192.168.18.3:443
--kubeconfig=/etc/kubelet/kubeconfig
```

最后重启kubelet服务。

5) 设置kube-proxy的启动参数

kube-proxy复用上一步kubelet创建的客户端证书，配置启动参数：

```
--master=https://192.168.18.3:443
--kubeconfig=/etc/kubernetes/kubeconfig
```

重启kube-proxy服务。

至此，一个基于CA的双向数字证书认证的Kubernetes集群环境就搭建完成了。

6) 设置kubectl客户端使用安全方式访问apiserver

在使用kubectl对Kubernetes集群进行操作时，默认使用非安全端口8080对apiserver进行访问，也可以设置为安全访问apiserver的模式，需要设置3个证书相关的参数“—certificate-authority”“--client-

certificate”和“--client-key”，分别表示用于CA授权的证书、客户端证书和客户端密钥。

- --certificate-authority: 使用为kube-apiserver生成的ca.crt文件。
- --client-certificate : 使用为 kube-controller-manager 生成的 cs_client.crt文件。
- --client-key: 使用为kube-controller-manager生成的cs_client.key文件。

同时，指定 apiserver 的 URL 地址为 HTTPS 安全地址（例如 https://k8s-master: 443），最后输入需要执行的子命令，即可对 apiserver 进行安全访问了：

```
# kubectl --server=https://k8s-master:443 --
certificate-authority=/etc/kubernetes/ssl_keys/ca.crt --
client-certificate=/etc/kubernetes/ssl_keys/cs_client.crt -
-client-key=/etc/kubernetes/ssl_keys/cs_client.key      get
nodes
```

NAME	STATUS	AGE
k8s-node-1	Ready	1h

2.基于HTTP BASE或TOKEN的简单认证方式

除了基于CA的双向数字证书认证方式，Kubernetes也提供了基于HTTP BASE或TOKEN的简单认证方式。各组件与apiserver之间的通信方式仍然采用HTTPS，但不使用CA数字证书。

采用基于HTTP BASE或TOKEN的简单认证方式时，API Server对外暴露HTTPS端口，客户端提供用户名、密码或Token来完成认证过程。需要说明的是，kubectl命令行工具比较特殊，它同时支持CA双向认证与简单认证两种模式与apiserver通信，其他客户端组件只能配置为双向安全认证或非安全模式与apiserver通信。

基于HTTP BASE认证的配置过程如下。

(1) 创建包括用户名、密码和UID的文件basic_auth_file，放置在合适的目录中，例如/etc/kubernetes目录。需要注意的是，这是一个纯文本文件，用户名、密码都是明文。

```
# vi /etc/kubernetes/basic_auth_file
admin,admin,1
system,system,2
```

(2) 设置kube-apiserver的启动参数“--basic_auth_file”，使用上述文件提供安全认证：

```
--secure-port=443
--basic_auth_file=/etc/kubernetes/basic_auth_file
```

然后，重启API Server服务。

(3) 使用kubectl通过指定的用户名和密码来访问API Server：

```
# kubectl --server=https://192.168.18.3:443 --
username=admin --password=admin --insecure-skip-tls-
```

```
verify=true get nodes
```

基于**TOKEN**认证的配置过程如下。

(1) 创建包括用户名、密码和UID的文件token_auth_file，放置在合适的目录中，例如/etc/kubernetes目录。需要注意的是，这是一个纯文本文件，用户名、密码都是明文。

```
$ cat /etc/kubernetes/token_auth_file
admin,admin,1
system,system,2
```

(2) 设置kube-apiserver的启动参数“--token_auth_file”，使用上述文件提供安全认证：

```
--secure-port=443
--token_auth_file=/etc/kubernetes/token_auth_file
```

然后，重启API Server服务。

(3) 用curl验证和访问API Server:

```
$ curl -k --header "Authorization:Bearer admin"
https://192.168.18.3:443/version
{
  "major": "1",
  "minor": "3",
  "gitVersion": "v1.3.3",
```

```

"gitCommit":
"c6411395e09da356c608896d3d9725acab821418",
  "gitTreeState": "clean",
  "buildDate": "2016-07-22T20:22:25Z",
  "goVersion": "go1.6.2",
  "compiler": "gc",
  "platform": "linux/amd64"
}

```

2.1.4 Kubernetes的版本升级

Kubernetes的版本升级需要考虑到当前集群中正在运行的容器不受影响。应对集群中的各Node逐个进行隔离，然后等待在其上运行的容器全部执行完成，再更新该Node上的kubelet和kube-proxy服务，将全部Node都更新完成后，最后更新Master的服务。

- 通过官网获取最新版本的二进制包kubernetes.tar.gz，解压缩后提取服务二进制文件。
- 逐个隔离Node，等待在其上运行的全部容器工作完成，更新kubelet和kube-proxy服务文件，然后重启这两个服务。
- 更新Master的kube-apiserver、kube-controller-manager、kube-scheduler服务文件并重启。

2.1.5 内网中的Kubernetes相关配置

Kubernetes在能够访问Internet网络的环境中使用起来非常方便，一方面在docker.io和gcr.io网站中已经存在了大量官方制作的Docker镜像，另一方面GCE、AWS提供的云平台已经很成熟了，用户通过租用一定的空间来部署Kubernetes集群也很简便。

但是，许多企业内部由于安全性的原因无法访问Internet。对于这些企业就需要通过创建一个内部的私有Docker Registry，并修改一些Kubernetes的配置，来启动内网中的Kubernetes集群。

1.Docker Private Registry（私有Docker镜像库）

使用Docker提供的Registry镜像创建一个私有镜像仓库。

详细的安装步骤请参考Docker的官方文档<https://docs.docker.com/registry/deploying/>。

2.kubelet配置

由于在Kubernetes中是以Pod而不是Docker容器为管理单元的，在kubelet创建Pod时，还通过启动一个名为google_containers/pause的镜像来实现Pod的概念。

该镜像存在于谷歌镜像库<http://gcr.io>中，需要通过一台能够连上Internet的服务器将其下载，导出文件，再push到私有Docker Registry中去。

之后，可以给每台Node的kubelet服务的启动参数加上--pod_infra_container_image参数，指定为私有Docker Registry中pause镜像的地址。例如：

```
# cat /etc/kubernetes/kubelet

KUBELET_ARGS="--api-servers=http://192.168.18.3:8080 -
-hostname-override=192.168.18.3                --log-
dir=/var/log/kubernetes                        --v=2
pod_infra_container_image=gcr.io/google_containers/pause-
amd64:3.0"
```

如果该镜像无法下载，则也可以从Docker Hub上进行下载：

```
# docker pull kubeguide/google_containers/pause-
amd64:3.0
```

修改kubelet配置文件中的—pod_infra_container_image参数如下：

```
--

pod_infra_container_image=kubeguide/google_containers/pause
-amd64:3.0
```

然后重启kubelet服务：

```
# systemctl restart kubelet
```

通过以上设置就在内网环境中搭建了一个企业内部的私有容器云平台。

2.1.6 Kubernetes核心服务配置详解

我们在2.1.2节对Kubernetes各服务启动进程的关键配置参数进行了简要说明，实际上Kubernetes的每个服务都提供了许多可配置的参数。这些参数涉及安全性、性能优化及功能扩展（Plugin）等方方面面。全面理解和掌握这些参数的含义和配置，无论对于Kubernetes的生产部署还是日常运维都有很好的帮助。

每个服务的可用参数都可以通过运行“`cmd--help`”命令进行查看，其中`cmd`为具体的服务启动命令，例如`kube-apiserver`、`kube-controller-manager`、`kube-scheduler`、`kubelet`、`kube-proxy`等。另外，也可以通过在命令的配置文件（例如`/etc/kubernetes/kubelet`等）中添加“`--参数名=参数取值`”的语句来完成对某个参数的配置。

本节将对Kubernetes所有服务的参数进行全面介绍，为了方便学习和查阅，对每个服务的参数用一个小节进行详细说明。

1.公共配置参数

公共配置参数适用于所有服务，如表2.3所示的参数可用于`kube-apiserver`、`kube-controller-manager`、`kube-scheduler`、`kubelet`、`kube-proxy`。本节对这些参数进行统一说明，不再在每个服务的参数列表中列出。

表2.3 公共配置参数表

参数名和取值示例	说 明
--log-backtrace-at=:0	记录日志每到“file:行号”时打印一次 stack trace
--log-dir=	日志文件路径
--log-flush-frequency=5s	设置 flush 日志文件的时间间隔
--logtostderr=true	设置为 true 则表示将日志输出到 stderr，不输出到日志文件

续表

参数名和取值示例	说 明
--alsologtostderr=false	设置为 true 则表示将日志输出到文件的同时输出到 stderr
--stderrthreshold=2	将该 threshold 级别之上的日志输出到 stderr
--v=0	glog 日志级别
--vmodule=	glog 基于模块的详细日志级别
--version=[false]	设置为 true 则将打印版本信息然后退出

2.kube-apiserver启动参数

对kube-apiserver启动参数的详细说明如表2.4所示。

表2.4 对kube-apiserver启动参数的详细说明

参数名和取值示例	说 明
--admission-control="AlwaysAdmit"	<p>对发送给 API Server 的任何请求进行准入控制，配置为一个“准入控制器”的列表，多个准入控制器时以逗号分隔。多个准入控制器将按顺序对发送给 API Server 的请求进行拦截和过滤，若某个准入控制器不允许该请求通过，则 API Server 拒绝此调用请求。可配置的准入控制器如下。</p> <ul style="list-style-type: none"> AlwaysAdmit: 允许所有请求。 AlwaysPullImages: 在启动容器之前总是去下载镜像，相当于在每个容器的配置项 imagePullPolicy=Always。 AlwaysDeny: 禁止所有请求，一般用于测试。 DenyExecOnPrivileged: 它会拦截所有想在 privileged container 上执行命令的请求。如果你的集群支持 privileged container，你又希望限制用户在这些 privileged container 上执行命令，那么强烈推荐你使用它。 ServiceAccount: 这个 plug-in 将 serviceAccounts 实现了自动化，如果你想要使用 ServiceAccount 对象，那么强烈推荐你使用它。 SecurityContextDeny: 这个插件将使用了 SecurityContext 的 Pod 中定义的选项全部失效。SecurityContext 在 container 中定义了操作系统级别的安全设定（uid、gid、capabilities、SELinux 等）。 ResourceQuota: 用于配额管理目的，作用于 Namespace 上，它会观察所有的请求，确保在 namespace 上的配额不会超标。推荐在 admission control 参数列表中这个插件排最后一个。 LimitRanger: 用于配额管理，作用于 Pod 与 Container 上，确保 Pod 与 Container 上的配额不会超标。 NamespaceExists（已过时）：对所有请求校验 namespace 是否存在，如果不存在则拒绝请求。已合并至 NamespaceLifecycle。 NamespaceAutoProvision（已过时）：对所有请求校验 namespace，如果不存在则自动创建该 namespace，推荐使用 NamespaceLifecycle。

续表

参数名和取值示例	说 明
<code>--admission-control="AlwaysAdmit"</code>	<ul style="list-style-type: none"> NamespaceLifecycle: 如果尝试在一个不存在的 namespace 中创建资源对象, 则该创建请求将被拒绝。当删除一个 namespace 时, 系统将会删除该 namespace 中的所有对象, 包括 Pod、Service 等。 如果启用多种准入选项, 则建议加载的顺序是: <code>--admission-control=NamespaceLifecycle,LimitRanger,SecurityContextDeny,ServiceAccount,ResourceQuota</code>
<code>--admission-control-config-file=""</code>	与准入控制规则相关的配置文件
<code>--advertise-address=<nil></code>	用于广播给集群的所有成员自己的 IP 地址, 不指定该地址将使用“ <code>--bind-address</code> ”定义的 IP 地址
<code>--allow-privileged=false</code>	如果设置为 true, 则 Kubernetes 将允许在 Pod 中运行拥有系统特权的容器应用, 与 <code>docker run --privileged</code> 的功效相同
<code>--apiserver-count=1</code>	集群中运行的 API Server 数量
<code>--authentication-token-webhook-cache-ttl=2m0s</code>	将 webhook token authenticator 返回的响应保存在缓存内的时间, 默认为两分钟
<code>--authentication-token-webhook-config-file=""</code>	Webhook 相关的配置文件, 将用于 token authentication
<code>--authorization-mode="AlwaysAllow"</code>	到 API Server 的安全访问的认证模式列表, 以逗号分隔, 可选值包括: AlwaysAllow、AlwaysDeny、ABAC、Webhook、RBAC
<code>--authorization-policy-file=""</code>	当 <code>--authorization-mode</code> 设置为 ABAC 时使用的 csv 格式的授权配置文件
<code>--authorization-rbac-super-user=""</code>	当 <code>--authorization-mode</code> 设置为 RBAC 时使用的超级用户名, 使用该用户名可以不进行 RBAC 认证
<code>--authorization-webhook-cache-authorized-ttl=5m0s</code>	将 webhook authorizer 返回的“已授权”响应保存在缓存内的时间, 默认为 5 分钟。
<code>--authorization-webhook-cache-unauthorized-ttl=30s</code>	将 webhook authorizer 返回的“未授权”响应保存在缓存内的时间, 默认为 30 秒
<code>--authorization-webhook-config-file=""</code>	当 <code>--authorization-mode</code> 设置为 webhook 时使用的授权配置文件
<code>--basic-auth-file=""</code>	设置该文件用于通过 HTTP 基本认证的方式访问 API Server 的安全端口
<code>--bind-address=0.0.0.0</code>	Kubernetes API Server 在本地地址的 6443 端口开启安全的 HTTPS 服务, 默认为 0.0.0.0
<code>--cert-dir="/var/run/kubernetes"</code>	TLS 证书所在的目录, 默认为 <code>/var/run/kubernetes</code> 。如果设置了 <code>--tls-cert-file</code> 和 <code>--tls-private-key-file</code> , 则该设置将被忽略
<code>--client-ca-file=""</code>	如果指定, 则该客户端证书将被用于认证过程
<code>--cloud-config=""</code>	云服务商的配置文件路径, 不配置则表示不使用云服务商的配置文件
<code>--cloud-provider=""</code>	云服务商的名称, 不配置则表示不使用云服务商
<code>--cors-allowed-origins=[]</code>	CORS (跨域资源共享) 设置允许访问的源域列表, 用逗号分隔, 并可使用正则表达式匹配子网。如果不指定, 则表示不启用 CORS
<code>--delete-collection-workers=1</code>	启动 DeleteCollection 的工作线程数, 用于提高清理 namespace 的效率
<code>--deserialization-cache-size=50000</code>	设置内存中缓存的 JSON 对象的个数

续表

参数名和取值示例	说 明
--enable-garbage-collector[=false]	设置为 true 表示启用垃圾回收器。必须与 kube-controller-manager 的该参数设置为相同的值
--enable-swagger-ui[=false]	设置为 true 表示启用 swagger ui 网页, 可通过 API Server 的 URL/swagger-ui 访问
--etcd-cafile=""	到 etcd 安全连接使用的 SSL CA 文件
--etcd-certfile=""	到 etcd 安全连接使用的 SSL 证书文件
--etcd-keyfile=""	到 etcd 安全连接使用的 SSL key 文件
--etcd-prefix="/registry"	在 etcd 中保存 Kubernetes 集群数据的根目录名, 默认为 /registry
--etcd-quorum-read[=false]	设置为 true 表示启用 quorum read 机制
--etcd-servers=[]	以逗号分隔的 etcd 服务 URL 列表, etcd 服务以 http://ip:port 格式表示
--etcd-servers-overrides=[]	按资源覆盖 etcd 服务的设置, 以逗号分隔。单个覆盖格式为: group/resource #servers, 其中 servers 格式为 http://ip:port, 以分号分隔
--event-ttl=1h0m0s	Kubernetes API Server 中各种事件 (通常用于审计和追踪) 在系统中保存的时间, 默认为 1 小时
--experimental-keystone-url=""	设置 keystone 鉴权插件地址, 实验用
--external-hostname=""	用于生成该 Master 的对外 URL 地址, 例如用于 swagger api 文档中的 URL 地址。
--insecure-bind-address=127.0.0.1	绑定的不安全 IP 地址, 与 --insecure-port 共同使用, 默认为 localhost。设置为 0.0.0.0 表示使用全部网络接口
--insecure-port=8080	提供非安全认证访问的监听端口, 默认为 8080。应在防火墙中进行配置, 以使得外部客户端不可以通过非安全端口访问 API Server
--ir-data-source="influxdb"	设置 InitialResources 使用的数据库, 可配置项包括 influxdb、gcm
--ir-database="k8s"	InitialResources 所需指标保存在 influxdb 中的数据库名称, 默认为 k8s
--ir-hawkular=""	设置 Hawkular 的 URL 地址
--ir-influxdb-host="localhost:8080/api/v1/proxy/namespaces/kube-system/services/monitoring-influxdb:api"	InitialResources 所需指标所在 influxdb 的 URL 地址, 默认为 localhost:8080/api/v1/proxy/namespaces/kube-system/services/monitoring-influxdb:api
--ir-namespace-only[=false]	设置为 true 表示从相同的 namespace 内的数据进行估算
--ir-password="root"	连接 influxdb 数据库的密码
--ir-percentile=90	InitialResources 进行资源估算时的采样百分比, 实验用
--ir-user="root"	连接 influxdb 数据库的用户名
--kubelet-certificate-authority=""	用于 CA 授权的 cert 文件路径
--kubelet-client-certificate=""	用于 TLS 的客户端证书文件路径
--kubelet-client-key=""	用于 TLS 的客户端 key 文件路径
--kubelet-https[=true]	指定 kubelet 是否使用 HTTPS 连接
--kubelet-timeout=5s	kubelet 执行操作的超时时间
--kubernetes-service-node-port=0	设置 Master 服务是否使用 NodePort 模式, 如果设置, 则 Master 服务将映射到物理机的端口号; 设置为 0 表示以 ClusterIP 的形式启动 Master 服务

续表

参数名和取值示例	说 明
--long-running-request-regexp="(\\^)((watch proxy)(/)?\$)(logs? portforward exec attach)/?\$)"	以正则表达式配置哪些需要长时间执行的请求不会被系统进行超时处理
--master-service-namespace="default"	设置 Master 服务所在的 namespace，默认为 default
--max-connection-bytes-per-sec=0	设置为非 0 的值表示限制每个客户端连接的带宽为 xx 字节/秒，目前仅用于需要长时间执行的请求
--max-requests-inflight=400	同时处理的最大请求数量，默认为 400，超过该数量的请求将被拒绝。设置为 0 表示无限制
--min-request-timeout=1800	最小请求处理超时时间，单位为秒，默认为 1800 秒，目前仅用于 watch request handler，其将会在该时间值上加一个随机时间作为请求的超时时间
--oidc-ca-file=""	该文件内设置鉴权机构，OpenID Server 的证书将被其中一个机构进行验证。如果不设置，则将使用主机的 root CA 证书
--oidc-client-id=""	OpenID Connect 的客户端 ID，在 oidc-issuer-url 设置时必须设置
--oidc-groups-claim=""	定制的 OpenID Connect 用户组声明的设置，以字符串数组的形式表示，实验用
--oidc-issuer-url=""	OpenID 发行者的 URL 地址，仅支持 HTTPS scheme，用于验证 OIDC JSON Web Token (JWT)
--oidc-username-claim="sub"	OpenID claim 的用户名，默认为 "sub"，实验用
--profiling=true	打开性能分析，可以通过 <host>:<port>/debug/pprof 地址查看程序栈、线程等系统信息
--repair-malformed-updates[=true]	设置为 true 表示服务器将尽可能修复无效或格式错误的 update request，以通过正确性校验，例如在一个 update request 中将一个已存在的 UID 值设置为空
--runtime-config=	一组 key=value 用于运行时的配置信息。apis/<groupVersion>/<resource> 可用于打开或关闭对某个 API 版本的支持。api/all 和 api/legacy 特别用于支持所有版本的 API 或支持旧版本的 API
--secure-port=6443	设置 API Server 使用的 HTTPS 安全模式端口号，设置为 0 表示不启用 HTTPS
--service-account-key-file=""	包含 PEM-encoded x509 RSA 公钥和私钥的文件路径，用于验证 Service Account 的 token。不指定则使用 --tls-private-key-file 指定的文件
--service-account-lookup[=false]	设置为 true 时，系统会到 etcd 验证 ServiceAccount token 是否存在
--service-cluster-ip-range=<nil>	Service 的 Cluster IP (虚拟 IP) 池，例如 169.169.0.0/16，这个 IP 地址池不能与物理机所在的网络重合
--service-node-port-range=	Service 的 NodePort 能使用的主机端口号范围，默认为 30000~32767，包括 30000 和 32767
--ssh-keyfile=""	如果指定，则通过 SSH 使用指定的秘钥文件对 Node 进行访问
--ssh-user=""	如果指定，则通过 SSH 使用指定的用户名对 Node 进行访问
--storage-backend=""	设置持久化存储类型，可选项为 etcd2 (默认)、etcd3
--storage-media-type="application/json"	持久化存储中的保存格式，默认为 application/json。某些资源类型只能使用 application/json 格式进行保存，将忽略这个参数的设置

续表

参数名和取值示例	说 明
--storage-versions="apps/v1alpha1,authentication.k8s.io/v1beta1,authorization.k8s.io/v1beta1,autoscaling/v1,batch/v1,componentconfig/v1alpha1,extensions/v1beta1,policy/v1alpha1,rbac.authorization.k8s.io/v1alpha1,v1"	持久化存储的资源版本号，以分组形式标记，例如"group1/version1,group2/version2,..."
--tls-cert-file=""	包含 x509 证书的文件路径，用于 HTTPS 认证
--tls-private-key-file=""	包含 x509 与 tls-cert-file 对应的私钥文件路径
--token-auth-file=""	用于访问 API Server 安全端口的 token 认证文件路径
--watch-cache[=true]	设置为 true 表示将 watch 进行缓存
--watch-cache-sizes=[]	设置各资源对象 watch 缓存大小的列表，以逗号分隔，每个资源对象的设置格式为 resource#size，当 watch-cache 设置为 true 时生效

3.kube-controller-manager启动参数

对kube-controller-manager启动参数的详细说明如表2.5所示。

表2.5 对kube-controller-manager启动参数的详细说明

参数名和取值示例	说 明
--address=0.0.0.0	监听的主机 IP 地址，默认为 0.0.0.0 表示使用全部网络接口
--allocate-node-cidrs=false	设置为 true 表示使用云服务商为 Pod 分配的 CIDRs，仅用于公有云
--cloud-config=""	云服务商的配置文件路径，仅用于公有云
--cloud-provider=""	云服务商的名称，仅用于公有云
--cluster-cidr=<nil>	集群中 Pod 的可用 CIDR 范围
--cluster-name="kubernetes"	集群的名称，默认为 kubernetes
--concurrent-deployment-syncs=5	设置允许的并发同步 deployment 对象的数量，值越大表示同步操作越快，但将会消耗更多的 CPU 和网络资源
--concurrent-endpoint-syncs=5	设置并发执行 Endpoint 同步操作的数量，值越大表示同步操作越快，但将会消耗更多的 CPU 和网络资源
--concurrent-rc-syncs=5	并发执行 RC 同步操作的协程数，值越大表示同步操作越快，但将会消耗更多的 CPU 和网络资源
--concurrent-namespace-syncs=2	设置允许的并发同步 namespace 对象的数量，值越大表示同步操作越快，但将会消耗更多的 CPU 和网络资源
--concurrent-rc-syncs=5	设置允许的并发同步 replication controller 对象的数量，值越大表示同步操作越快，但将会消耗更多的 CPU 和网络资源
--concurrent-replicaset-syncs=5	设置允许的并发同步 replica set 对象的数量，值越大表示同步操作越快，但将会消耗更多的 CPU 和网络资源

续表

参数名和取值示例	说 明
--concurrent-resource-quota-syncs=5	设置允许的并发同步 resource quota 对象的数量, 值越大表示更快地进行同步操作, 但将会消耗更多的 CPU 和网络资源
--configure-cloud-routes[=true]	设置为 true 表示使用 allocate-node-cidr 进行 CIDRs 的分配, 仅用于公有云
--controller-start-interval=0	启动各个 controller manager 的时间间隔, 默认为 0 秒
--daemonset-lookup-cache-size=1024	DaemonSet 的查询缓存大小, 默认为 1024。值越大表示 DaemonSet 响应越快, 内存消耗也越大
--deleting-pods-burst=10	如果一个 Node 节点失败, 则会批量删除在上面运行的 Pod 实例的信息, 此值定义了突发最大删除的 Pod 的数量, 与 deleting-pods-qps 一起作为调度中的限流因子
--deleting-pods-qps=0.1	当 Node 失效时, 每秒删除其上的多少个 Pod 实例
--deployment-controller-sync-period=30s	同步 deployments 的时间间隔, 默认为 30 秒
--enable-dynamic-provisioning[=true]	设置为 true 表示启用动态 provisioning (需环境支持)
--enable-garbage-collector[=false]	设置为 true 表示启用垃圾回收机制, 必须与 kube-apiserver 的该参数设置为相同的值
--enable-hostpath-provisioner[=false]	设置为 true 表示启用 hostPath PV provisioning 机制, 仅用于测试, 不可用于多 Node 的集群环境
--flex-volume-plugin-dir="/usr/libexec/kubernetes/kubelet-plugins/volume/exec/"	设置 flex volume 插件应搜索其他第三方 volume 插件的全路径
--horizontal-pod-autoscaler-sync-period=30s	Pod 自动扩容器的 Pod 数量的同步时间间隔, 默认为 30 秒
--kube-api-burst=30	发送到 API Server 的每秒的请求数量, 默认为 30
--kube-api-content-type="application/vnd.kubernetes.protobuf"	发送到 API Server 的请求内容类型
--kube-api-qps=20	与 API Server 通信的 QPS 值, 默认为 20
--kubeconfig=""	kubeconfig 配置文件路径, 在配置文件中包括 Master 地址信息及必要的认证信息
--leader-elect[=false]	设置为 true 表示进行 leader 选举, 用于多个 Master 组件的高可用部署
--leader-elect-lease-duration=15s	leader 选举过程中非 leader 等待选举的时间间隔, 默认为 15 秒, 当 leader-elect=true 时生效
--leader-elect-renew-deadline=10s	leader 选举过程中在停止 leading 角色之前再次 renew 的时间间隔, 应小于或等于 leader-elect-lease-duration, 默认为 10 秒, 当 leader-elect=true 时生效
--leader-elect-retry-period=2s	leader 选举过程中在获取 leader 角色和 renew 之间的等待时间, 默认为两秒, 当 leader-elect=true 时生效
--master=""	API Server 的 URL 地址, 设置后不再使用 kubeconfig 中设置的值
--min-resync-period=12h0m0s	最小重新同步的时间间隔, 实际重新同步的时间为 MinResyncPeriod (默认为 12 小时) 到 2×MinResyncPeriod (默认 24 小时) 之间的一个随机数
--namespace-sync-period=5m0s	namespace 生命周期更新的同步时间间隔, 默认为 5 分钟
--node-cidr-mask-size=24	Node CIDR 的子网掩码设置, 默认为 24

续表

参数名和取值示例	说 明
--node-monitor-grace-period=40s	监控 Node 状态的时间间隔，默认为 40 秒，超过该设置时间后，controller-manager 会把 Node 标记为不可用状态。此值的设置有如下要求： 它应该被设置为 kubelet 汇报的 Node 状态时间间隔（参数—node-status-update-frequency=10s）的 N 倍， N 为 kubelet 状态汇报的重试次数
--node-monitor-period=5s	同步 NodeStatus 的时间间隔，默认为 5 秒
--node-startup-grace-period=1m0s	Node 启动的最大允许时间，超过此时间无响应则会标记 Node 为不可用状态（启动失败），默认为 1 分钟
--node-sync-period=10s	Node 信息发生变化时（例如新 Node 加入集群）controller-manager 同步各 Node 信息的时间间隔，默认为 10 秒
--pod-eviction-timeout=5m0s	在发现一个 Node 失效以后，延迟一段时间，在超过这个参数指定的时间后，删除此 Node 上的 Pod，默认为 5 分钟
--port=10252	controller-manager 监听的主机端口号，默认为 10252
--profiling=true	打开性能分析，可以通过<host>:<port>/debug/pprof/地址查看程序栈、线程等系统运行信息
--pv-recycler-increment-timeout-nfs=30	使用 nfs scrubber 的 Pod 每增加 1Gi 空间在 ActiveDeadlineSeconds 上增加的时间，默认为 30 秒
--pv-recycler-minimum-timeout-hostpath=60	使用 hostPath recycler 的 Pod 的最小 ActiveDeadlineSeconds 秒数，默认为 60 秒。实验用
--pv-recycler-minimum-timeout-nfs=300	使用 nfs recycler 的 Pod 的最小 ActiveDeadlineSeconds 秒数，默认为 300 秒
--pv-recycler-pod-template-filepath-hostpath=""	使用 hostPath recycler 的 Pod 的模板文件全路径，仅用于实验
--pv-recycler-pod-template-filepath-nfs=""	使用 nfs recycler 的 Pod 的模板文件全路径
--pv-recycler-timeout-increment-hostpath=30	使用 hostPath scrubber 的 Pod 每增加 1Gi 空间在 ActiveDeadlineSeconds 上增加的时间，默认为 30 秒。实验用
--pvclaimbinder-sync-period=15s	同步 PV 和 PVC（容器声明的 PV）的时间间隔
--replicaset-lookup-cache-size=4096	设置 replica sets 查询缓存的大小，默认为 4096，值越大表示查询操作越快，但将会消耗更多的内存
--replication-controller-lookup-cache-size=4096	设置 replication controller 查询缓存的大小，默认为 4096，值越大表示查询操作越快，但将会消耗更多的内存
--resource-quota-sync-period=5m0s	resource quota 使用信息同步的时间间隔，默认为 5 分钟
--root-ca-file=""	根 CA 证书文件路径，将被用于 Service Account 的 token secret 中
--service-account-private-key-file=""	用于给 Service Account token 签名的 PEM-encoded RSA 私钥文件路径
--service-cluster-ip-range=""	Service 的 IP 范围
--service-sync-period=5m0s	同步 service 与外部 load balancer 的时间间隔，默认为 5 分钟
--terminated-pod-gc-threshold=12500	设置可保存的终止 Pod 的数量，超过该数量，垃圾回收器将开始进行删除操作。设置为不大于 0 的值表示不启用该功能

4.kube-scheduler启动参数

对kube-scheduler启动参数的详细说明如表2.6所示。

表2.6 对kube-scheduler启动参数的详细说明

参数名和取值示例	说 明
--address=0.0.0.0	监听的主机 IP 地址，默认为 0.0.0.0 表示使用全部网络接口
--algorithm-provider="DefaultProvider"	设置调度算法，默认为 DefaultProvider
--failure-domains="kubernetes.io/hostname,failure-domain.beta.kubernetes.io/zone,failure-domain.beta.kubernetes.io/region"	表示 Pod 调度时的亲和力参数。在调度 Pod 时，如果两个 Pod 有相同的亲和力参数，那么这两个 Pod 会被调度到相同的 Node 上；如果两个 Pod 有不同的亲和力参数，那么这两个 Pod 不会被调度到相同的 Node 上
--hard-pod-affinity-symmetric-weight=1	表示 Pod 调度规则亲和力的权重值，取值范围为 0~100。RequiredDuringScheduling 亲和性是非对称的，但对每一个 RequiredDuringScheduling 亲和性都存在一个对应的隐式 PreferredDuringScheduling 亲和性规则。该设置表示隐式 PreferredDuringScheduling 亲和性规则的权重值，默认为 1
--kube-api-burst=100	发送到 API Server 的每秒请求数量，默认为 100
--kube-api-content-type="application/vnd.kubernetes.protobuf"	发送到 API Server 的请求内容类型
--kube-api-qps=50	与 API Server 通信的 QPS 值，默认为 50
--kubeconfig=""	kubeconfig 配置文件路径，在配置文件中包括 Master 的地址信息及必要的认证信息
--leader-elect[=false]	设置为 true 表示进行 leader 选举，用于多个 Master 组件的高可用部署
--leader-elect-lease-duration=15s	leader 选举过程中非 leader 等待选举的时间间隔，默认为 15 秒，当 leader-elect=true 时生效
--leader-elect-renew-deadline=10s	leader 选举过程中在停止 leading 角色之前再次 renew 的时间间隔，应小于或等于 leader-elect-lease-duration，默认为 10 秒，当 leader-elect=true 时生效
--leader-elect-retry-period=2s	leader 选举过程中获取 leader 角色和 renew 之间的等待时间，默认为两秒，当 leader-elect=true 时生效
--master=""	API Server 的 URL 地址，设置后不再使用 kubeconfig 中设置的值
--policy-config-file=""	调度策略（scheduler policy）配置文件的路径
--port=10251	scheduler 监听的主机端口号，默认为 10251
--profiling=true	打开性能分析，可以通过 <host>:<port>/debug/pprof/地址查看栈、线程等系统运行信息
--scheduler-name="default-scheduler"	调度器名称，用于选择哪些 Pod 将被该调度器进行处理，选择的依据是 Pod 的 annotation 设置，包含 key='scheduler.alpha.kubernetes.io/name'的 annotation

5.kubelet启动参数

对kubelet启动参数的详细说明如表2.7所示。

表2.7 对kubelet启动参数的详细说明

参数名和取值示例	说 明
--address=0.0.0.0	绑定主机 IP 地址，默认为 0.0.0.0 表示使用全部网络接口
--allow-privileged[=false]	是否允许以特权模式启动容器，默认为 false
--api-servers=[]	API Server 地址列表，以 ip:port 格式表示，以逗号分隔
--application-metrics-count-limit=100	为每个容器保存的性能指标的最大数量，默认为 100
--boot-id-file="/proc/sys/kernel/random/ boot_id"	以逗号分隔的文件列表，使用第 1 个存在 boot-id 的文件
--cadvisor-port=4194	本地 cAdvisor 监听的端口号，默认为 4194
--cert-dir="/var/run/kubernetes"	TLS 证书所在的目录，默认为 /var/run/kubernetes。如果设置了 --tls-cert-file 和 --tls-private-key-file，则该设置将被忽略
--cgroup-root=""	为 pods 设置的 root cgroup，如果不设置，则将使用容器运行时的默认设置
--chaos-chance=0	随机产生客户端错误的概率，仅用于测试，默认为 0，即不产生
--cloud-config=""	云服务商的配置文件路径
--cloud-provider="auto-detect"	云服务商的名称，默认将自动检测，设置为空表示无云服务商
--cluster-dns=""	集群内 DNS 服务的 IP 地址
--cluster-domain=""	集群内 DNS 服务所用域名
--config=""	kubelet 配置文件的路径或目录名
--configure-cbr0[=false]	设置为 true 表示 kubelet 将会根据 Node.Spec.PodCIDR 的值来配置 cbr0
--container-hints="/etc/cadvisor/container _hints.json"	容器 hints 文件所在的全路径
--container-runtime="docker"	容器类型，目前支持 Docker、rkt，默认为 docker
--containerized[=false]	将 kubelet 运行在容器中，仅供测试使用，默认为 false
--cpu-cfs-quota[=true]	设置为 true 表示启用 CPU CFS quota，用于设置容器的 CPU 限制
--docker-endpoint= "unix:///var/run/docker.sock"	Docker 服务的 Endpoint 地址，默认为 unix:///var/run/docker.sock
--docker-env-metadata-whitelist=""	Docker 容器需要使用的环境变量 key 列表，以逗号分隔
--docker-exec-handler="native"	进入 Docker 容器中执行命令的方式，支持 native、nsenter，默认为 native
--docker-only[=false]	设置为 true，表示仅报告 Docker 容器的统计信息而不再报告其他统计信息
--docker-root="/var/lib/docker"	Docker 根目录的全路径，不再使用，将通过 docker info 获取该信息
--enable-controller-attach-detach[=true]	设置为 true 表示启用 Attach/Detach Controller 进行调度到该 Node 的 volume 的 attach 与 detach 操作，同时禁用 kubelet 执行 attach、detach 的操作
--enable-custom-metrics[=false]	设置为 true 表示启用采集自定义性能指标
--enable-debugging-handlers=false	设置为 true 表示提供远程访问本节点容器的日志、进入容器执行命令等相关 Rest 服务
--enable-load-reader[=false]	设置为 true 表示启用 CPU 负载的 reader
--enable-server[=true]	启动 kubelet 上的 http rest server，此 server 提供了获取本节点上运行的 Pod 列表、Pod 状态和其他管理监控相关的 Rest 接口

续表

参数名和取值示例	说 明
--event-burst=10	临时允许的 Event 记录突发的最大数量，默认为 10，当 event-qps>0 时生效
--event-qps=5	设置大于 0 的值表示限制每秒能创建出的 Event 数量，设置为 0 表示不限制
--event-storage-age-limit="default=0"	保存 Event 的最大时间。按事件类型以 key=value 的格式表示，以逗号分隔，事件类型包括 creation、oom 等，“default”表示所有事件的类型
--event-storage-event-limit="default=0"	保存 Event 的最大数量。按事件类型以 key=value 格式表示，以逗号分隔，事件类型包括 creation、oom 等，“default”表示所有事件的类型
--eviction-hard=""	触发 Pod Eviction 操作的一组硬门限设置，例如可用内存<1Gi
--eviction-max-pod-grace-period=0	终止 Pod 操作给 Pod 自行停止预留的时间，单位为秒。时间到达时，将触发 Pod Eviction 操作。默认值为 0，设置为负数表示使用 Pod 中指定的值
--eviction-pressure-transition-period=5m0s	kubelet 在触发 Pod Eviction 操作之前等待的最长时间，默认为 5 分钟
--eviction-soft=""	触发 Pod Eviction 操作的一组软门限设置，例如可用内存<1.5Gi，与 grace-period 一起生效，当 Pod 的响应时间超过 grace-period 后进行触发
--eviction-soft-grace-period=""	触发 Pod Eviction 操作的一组软门限等待时间设置，例如 memory.available=1m30s
--exit-on-lock-contention[=false]	设置为 true 表示当有文件锁存在时 kubelet 也可以退出
--experimental-flannel-overlay[=false]	实验性功能，用于 kubelet 启动时自动支持 flannel 覆盖网络，默认值为 false
--experimental-nvidia-gpus=0	本节点上 NVIDIA GPU 的数量，目前仅支持 0 或 1，默认为 0
--file-check-frequency=20s	在 File Source 作为 Pod 源的情况下，kubelet 定期重新检查文件变化的时间间隔，文件发生变化后，kubelet 重新加载更新的文件内容
--global-housekeeping-interval=1m0s	全局 housekeeping 的时间间隔，默认为 1 分钟
--google-json-key=""	用于谷歌的云平台 Service Account 进行用于鉴权的 JSON key
--hairpin-mode="promiscuous-bridge"	设置 hairpin 模式，表示 kubelet 设置 hairpin NAT 的方式。该模式允许后端 Endpoint 在访问其本身 Service 时能够再次 loadbalance 回自身。可选项包括 promiscuous-bridge、hairpin-veth 和 none
--healthz-bind-address=127.0.0.1	healthz 服务监听的 IP 地址，默认为 127.0.0.1，设置为 0.0.0.0 表示监听全部 IP 地址
--healthz-port=10248	本地 healthz 服务监听的端口号，默认为 10248
--host-ipc-sources="*"	kubelet 允许 Pod 使用宿主机 ipc namespace 的列表，以逗号分隔，默认为 "*"
--host-network-sources="*"	kubelet 允许 Pod 使用宿主机 network 的列表，以逗号分隔，默认为 "*"
--host-pid-sources="*"	kubelet 允许 Pod 使用宿主机 pid namespace 的列表，以逗号分隔，默认为 "*"
--hostname-override=""	设置本 Node 在集群中的主机名，不设置将使用本机 hostname
--housekeeping-interval=10s	对容器做 housekeeping 操作的时间间隔，默认为 10 秒
--http-check-frequency=20s	HTTP URL Source 作为 Pod 源的情况下，kubelet 定期检查 URL 返回的内容是否发生变化的时间周期，作用同 file-check-frequency 参数
--image-gc-high-threshold=90	镜像垃圾回收上限，磁盘使用空间达到该百分比时，镜像垃圾回收将持续工作
--image-gc-low-threshold=80	镜像垃圾回收下限，磁盘使用空间在达到该百分比之前，镜像垃圾回收将不启用
--kube-api-burst=10	发送到 API Server 的每秒请求数量，默认为 10

续表

参数名和取值示例	说 明
--kube-api-content-type="application/vnd.kubernetes.protobuf"	发送到 API Server 的请求内容类型
--kube-api-qps=5	与 API Server 通信的 QPS 值，默认为 5
--kube-reserved=	kubernetes 系统预留的资源配置，以一组 ResourceName=ResourceQuantity 格式表示，例如 cpu=200m,memory=150G。目前仅支持 CPU 和内存的设置，详见 http://releases.k8s.io/HEAD/docs/user-guide/compute-resources.md ，默认为空
--kubeconfig="/var/lib/kubelet/kubeconfig"	kubeconfig 配置文件路径，在配置文件中包括 Master 地址信息及必要的认证信息
--kubelet-cgroups=""	kubelet 运行所需的 cgroups 名称
--lock-file=""	kubelet 使用的 lock 文件，Alpha 版本
--log-cadvisor-usage[=false]	设置为 true 表示将 cAdvisor 容器的使用情况进行日志记录
--low-diskspace-threshold-mb=256	本 Node 最低磁盘可用空间，单位 MB。当磁盘空间低于该阈值，kubelet 将拒绝创建新的 Pod，默认值为 256MB
--machine-id-file="/etc/machine-id,/var/lib/dbus/machine-id"	用于查找 machine-id 的文件列表，使用找到的第 1 个值，默认从/etc/machine-id,/var/lib/dbus/machine-id 文件中去找
--manifest-uri=""	为 HTTP URL Source 源类型时，kubelet 用来获取 Pod 定义的 URL 地址，此 URL 返回一组 Pod 定义
--manifest-url-header=""	访问 manifest URL 地址时使用的 HTTP 头信息，以 key=value 格式表示
--master-service-namespace="default"	Master 服务的命名空间，默认为 default
--max-open-files=1000000	kubelet 打开的最大文件数量，默认为 1000 000
--max-pods=110	kubelet 能运行的最大 Pod 数量，默认为 110 个 Pod
--maximum-dead-containers=240	在本 Node 上保留的已停止容器的最大数量，由于停止的容器也会消耗磁盘空间，所以超过该上限以后，kubelet 会自动清理已停止的容器以释放磁盘空间，默认为 240
--maximum-dead-containers-per-container=2	以 Pod 为单位可以保留的已停止的（属于同一 Pod 的）容器集的最大数量
--minimum-container-ttl-duration=1m0s	已停止的容器在被清理之前的最小存活时间，例如 300ms、10s 或 2h45m，超过此存活时间的容器将被标记为可被 GC 清理，默认值为 1 分钟
--minimum-image-ttl-duration=2m0s	不再使用的镜像在被清理之前的最小存活时间，例如 300ms、10s 或 2h45m，超过此存活时间的镜像被标记为可被 GC 清理，默认值为两分钟
--network-plugin=""	自定义的网络插件的名字，Pod 的生命周期中相关的一些事件会调用此网络插件进行处理，为 Alpha 测试版功能
--network-plugin-dir="/usr/libexec/kubernetes/kubelet-plugins/net/exec/"	扫描网络插件的目录，为 Alpha 测试版功能
--node-ip=""	设置本 Node 的 IP 地址
--node-labels=	kubelet 注册本 Node 时设置的 Labels，label 以 key=value 的格式表示，多个 label 以逗号分隔，为 Alpha 测试版功能
--node-status-update-frequency=10s	kubelet 向 Master 汇报 Node 状态的时间间隔，默认值为 10 秒。与 controller-manager 的--node-monitor-grace-period 参数共同起作用
--non-masquerade-cidr="10.0.0.0/8"	kubelet 向该 IP 段之外的 IP 地址发送的流量将使用 IP Masquerade 技术

续表

参数名和取值示例	说 明
--oom-score-adj=-999	kubelet 进程的 oom_score_adj 参数值, 有效范围为[-1000, 1000]
--outofdisk-transition-frequency=5m0s	触发磁盘空间耗尽操作之前的等待时间, 默认为 5 分钟
--pod-cidr=""	用于给 Pod 分配 IP 地址的 CIDR 地址池, 仅在单机模式中使用。在一个集群中, kubelet 会从 API Server 中获取 CIDR 设置
--pod-infra-container-image="gcr.io/google_containers/pause-amd64:3.0"	用于 Pod 内网络命名空间共享的基础 pause 镜像
--pods-per-core=0	该 kubelet 上每个 core 可运行的 Pod 数量。最大值将被 max-pods 参数限制。默认值为 0 表示不做限制
--port=10250	kubelet 服务监听的本机端口号, 默认为 10250
--read-only-port=10255	kubelet 服务监听的“只读”端口号, 默认为 10255, 设置为 0 表示不启用
--really-crash-for-testing=false	设置为 true 表示发生 panics 情况时崩溃, 仅用于测试
--reconcile-cidr[=true]	根据 API Server 指定的 CIDR 重排 Node 的 CIDR 地址, 如果 register-node 或 configure-cbr0 设置为 false, 则表示不启用。默认值为 true
--register-node[=true]	将本 Node 注册到 API Server, 默认值为 true
--register-schedulable[=true]	将本 Node 状态标记为 schedulable, 设置为 false 表示通知 Master 本 Node 不可进行调度。默认值为 true
--registry-burst=10	最多同时拉取镜像的数量, 默认值为 10
--registry-qps=5	在 Pod 创建过程中容器的镜像可能需要从 Registry 中拉取, 由于拉取镜像的过程中会消耗大量带宽, 因此可能需要限速, 此参数与 registry-burst 一起用来限制每秒拉取多少个镜像, 默认不限速, 如果设置为 5, 则表示平均每秒允许拉取 5 个镜像
--resolv-conf="/etc/resolv.conf"	命名服务配置文件, 用于容器内应用的 DNS 解析, 默认为/etc/resolv.conf
--rkt-api-endpoint="localhost:15441"	rkt API 服务的 URL 地址, --container-runtime='rkt' 时生效
--rkt-path=""	rkt 二进制文件的路径, 不指定的话从环境变量 \$PATH 中查找, --container-runtime='rkt' 时生效
--root-dir="/var/lib/kubelet"	kubelet 运行根目录, 将保持 Pod 和 volume 的相关文件, 默认为/var/lib/kubelet。
--runonce=false	设置为 true 表示创建完 Pod 之后立即退出 kubelet 进程, 与--api-servers 和 --enable-server 参数互斥
--runtime-cgroups=""	为容器 runtime 设置的 cgroup
--runtime-request-timeout=2m0s	除了长时间运行的 request, 对其他 request 的超时时间设置, 包括 pull、logs、exec、attach 等操作。当超时时间到达时, 请求会被杀掉, 抛出一个错误并会重试。默认值为两分钟
--seccomp-profile-root="/var/lib/kubelet/seccomp"	seccomp 配置文件目录, 默认为/var/lib/kubelet/seccomp
--serialize-image-pulls[=true]	按顺序挨个 pull 镜像。建议 Docker 低于<1.9 版本或使用 aufs storage backend 时设置为 true, 详见 issue #10959
--storage-driver-buffer-duration=1m0s	将缓存数据写入后端存储的时间间隔, 默认为 1 分钟

续表

参数名和取值示例	说 明
--storage-driver-db="cadvisor"	后端存储的数据库名称，默认为 cadvisor
--storage-driver-host="localhost:8086"	后端存储的数据库连接 URL 地址，默认为 localhost:8086
--storage-driver-password="root"	后端存储的数据库密码，默认为 root
--storage-driver-secure[=false]	后端存储的数据库是否用安全连接，默认为 false
--storage-driver-table="stats"	后端存储的数据库表名，默认为 stats
--storage-driver-user="root"	后端存储的数据库用户名，默认为 root
--streaming-connection-idle-timeout=4h0m0s	在容器中执行命令或者进行端口转发的过程中会产生输入、输出流，这个参数用来控制连接空闲超时而关闭的时间，如果设置为“5m”，则表示连接超过 5 分钟没有输入、输出的情况下就被认为是空闲的，而会被自动关闭。默认为 4 小时
--sync-frequency=1m0s	同步运行中容器的配置的频率，默认为 1 分钟
--system-cgroups=""	kubelet 为运行非 kernel 进程设置的 cgroups 名称
--system-reserved=	系统预留的资源配置，以一组 ResourceName=ResourceQuantity 格式表示，例如 cpu=200m,memory=150G。目前仅支持 CPU 和内存的设置，详见 http://releases.k8s.io/HEAD/docs/user-guide/compute-resources.md ，默认为空
--tls-cert-file=""	包含 x509 证书的文件路径，用于 HTTPS 认证
--tls-private-key-file=""	包含 x509 与 tls-cert-file 对应的私钥文件路径
--volume-plugin-dir="/usr/libexec/kubernet etes/kubelet-plugins/volume/exec/"	搜索第三方 volume 插件的目录，为 Alpha 测试版功能
--volume-stats-aggr-period=1m0s	kubelet 计算所有 Pod 和 volume 的磁盘使用情况聚合值的时间间隔，默认为 1 分钟。设置为 0 表示不启用该计算功能

6.kube-proxy启动参数

kube-proxy的启动参数详细说明见表2.8。

表2.8 kube-proxy的参数表

参数名和取值示例	说 明
--bind-address=0.0.0.0	kube-proxy 绑定主机的 IP 地址，默认为 0.0.0.0 表示绑定所有 IP 地址
--cleanup-iptables[=false]	设置为 true 表示清除 iptables 规则后退出
--cluster-cidr=""	集群中 Pod 的 CIDR 地址范围，用于桥接集群外部流量到内部。用于公有云环境
--config-sync-period=15m0s	从 API Server 更新配置的时间间隔，默认为 15 分钟，必须大于 0
--conntrack-max=0	跟踪 NAT 连接的最大数量，默认值为 0 表示不限制
--conntrack-max-per-core=32768	跟踪每个 CPU core 的 NAT 连接的最大数量，默认值为 32768，仅当 conntrack-max 设置为 0 时生效
--conntrack-tcp-timeout-established=24h0m0s	建立 TCP 连接的超时时间，默认为 24 小时，设置为 0 表示无限制

续表

参数名和取值示例	说 明
--healthz-bind-address=127.0.0.1	healthz 服务绑定主机 IP 地址，默认为 127.0.0.1，设置为 0.0.0.0 表示使用所有 IP 地址
--healthz-port=10249	healthz 服务监听的主机端口号，默认为 10249
--hostname-override=""	设置本 Node 在集群中的主机名，不设置将使用本机 hostname
--iptables-masquerade-bit=14	iptables masquerade 的 fwmark 位设置，有效范围为[0, 31]
--iptables-sync-period=30s	刷新 iptables 规则的时间间隔，例如 5s、1m、2h22m，默认为 30 秒，必须大于 0
--kube-api-burst=10	发送到 API Server 的每秒发请求数量，默认为 10
--kube-api-content-type="application/vnd.kubernetes.protobuf"	发送到 API Server 的请求内容类型
--kube-api-qps=5	与 API Server 通信的 QPS 值，默认为 5
--kubeconfig=""	kubeconfig 配置文件路径，在配置文件中包括 Master 地址信息及必要的认证信息
--masquerade-all[=false]	设置为 true 表示使用纯 iptables 代理，所有网络包都将做 SNAT 转换
--master=""	API Server 的地址
--oom-score-adj=-999	kube-proxy 进程的 oom_score_adj 参数值，有效范围为[-1000,1000]
--proxy-mode=	代理模式，可选项为 iptables 或 userspace，默认为 iptables，转发速度更快。当操作系统 kernel 版本或 iptables 版本不够新时，将自动降级为 userspace 模式
--proxy-port-range=	进行 Service 代理的本地端口号范围，格式为 begin-end，含两端，未指定则采用随机选择的系统可用的端口号
--udp-timeout=250ms	保持空闲 UDP 连接的时间，例如 250ms、2s，默认值为 250ms，必须大于 0，仅当 proxy-mode=userspace 时生效

2.1.7 Kubernetes集群网络配置方案

在多个Node组成的Kubernetes集群内，跨主机的容器间网络互通是Kubernetes集群能够正常工作的前提条件。Kubernetes本身并不会对跨主机容器网络进行设置，这需要额外的工具来实现。除了谷歌公有云GCE平台提供的网络设置，一些开源的工具包括flannel、Open vSwitch、Weave、Calico等都能够实现跨主机的容器间网络互通。本节将对常用的flannel、Open vSwitch和直接路由三种配置进行详细说明。

1.flannel（覆盖网络）

flannel采用覆盖网络（Overlay Network）模型来完成对网络的打通，本节对flannel的安装和配置进行详细说明。

1) 安装etcd

由于flannel使用etcd作为数据库，所以需要预先安装好etcd，详见2.1.2节的说明。

2) 安装flannel

需要在每台Node上都安装flannel。flannel软件的下载地址为<https://github.com/coreos/flannel/releases>。将下载的压缩包flannel-`<version>`-linux-amd64.tar.gz解压，把二进制文件flanneld和mk-docker-

opts.sh复制到/usr/bin（或其他PATH环境变量中的目录），即可完成对flannel的安装。

3) 配置flannel

此处以使用systemd系统为例对flanneld服务进行配置。编辑服务配置文件/usr/lib/systemd/system/flanneld.service:

```
[Unit]
Description=flanneld overlay address etcd agent
After=network.target
Before=docker.service

[Service]
Type=notify
EnvironmentFile=/etc/sysconfig/flanneld
                                ExecStart=/usr/bin/flanneld      -etcd-
                                endpoints=${FLANNEL_ETCD} $FLANNEL_OPTIONS

[Install]
RequiredBy=docker.service
WantedBy=multi-user.target
```

编辑配置文件/etc/sysconfig/flannel，设置etcd的URL地址:

```
# flanneld configuration options

# etcd url location.  Point this to the server where
```

```
etcd runs

FLANNEL_ETCD="http://192.168.18.3:2379"

# etcd config key. This is the configuration key that
flannel queries

# For address range assignment

FLANNEL_ETCD_KEY="/coreos.com/network"
```

在启动flanneld服务之前，需要在etcd中添加一条网络配置记录，这个配置将用于flanneld分配给每个Docker的虚拟IP地址段。

```
# etcdctl set /coreos.com/network/config '{ "Network":
"10.1.0.0/16"}'
```

4) 由于flannel将覆盖docker0网桥，所以如果Docker服务已启动，则停止Docker服务。

5) 启动flanneld服务:

```
# systemctl restart flanneld
```

6) 设置docker0网桥的IP地址:

```
# mk-docker-opts.sh -i
# source /run/flannel/subnet.env
# ifconfig docker0 ${FLANNEL_SUBNET}
```

完成后确认网络接口docker0的IP地址属于flannel0的子网:

```
# ip addr

                                flannel0:
flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST>      mtu
1472
                                inet 10.1.10.0    netmask 255.255.0.0
destination 10.1.10.0
                                docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>
mtu 1500
                                inet 10.1.10.1    netmask 255.255.255.0
broadcast 10.1.10.255
```

7) 重新启动Docker服务:

```
# systemctl restart docker
```

到此就完成了flannel覆盖网络的设置。

使用ping命令验证各Node上docker0之间的相互访问。例如在Node1（docker0IP=10.1.10.1）机器上ping Node2的docker0（docker0's IP=10.1.30.1），通过flannel能够成功连接到其他物理机的Docker网络:

```
$ ping 10.1.30.1
PING10.1.30.1 (10.1.30.1) 56(84) bytes of data.
 64 bytes from 10.1.30.1: icmp_seq=1 ttl=62 time=1.15
ms
 64 bytes from 10.1.30.1: icmp_seq=2 ttl=62 time=1.16
```

ms

64 bytes from 10.1.30.1: icmp_seq=3 ttl=62 time=1.57

ms

我们也可以在etcd中查看到flannel设置的flannel0地址与物理机IP地址的对应规则:

```
# etcdctl ls /coreos.com/network/subnets
/coreos.com/network/subnets/10.1.10.0-24
/coreos.com/network/subnets/10.1.20.0-24
/coreos.com/network/subnets/10.1.30.0-24

# etcdctl get /coreos.com/network/subnets/10.1.10.0-24
{"PublicIP": "192.168.1.129"}
# etcdctl get /coreos.com/network/subnets/10.1.20.0-24
{"PublicIP": "192.168.1.130"}
# etcdctl get /coreos.com/network/subnets/10.1.30.0-24
{"PublicIP": "192.168.1.131"}
```

2.Open vSwitch（虚拟交换机）

以两个Node为例，目标网络拓扑如图2.2所示。

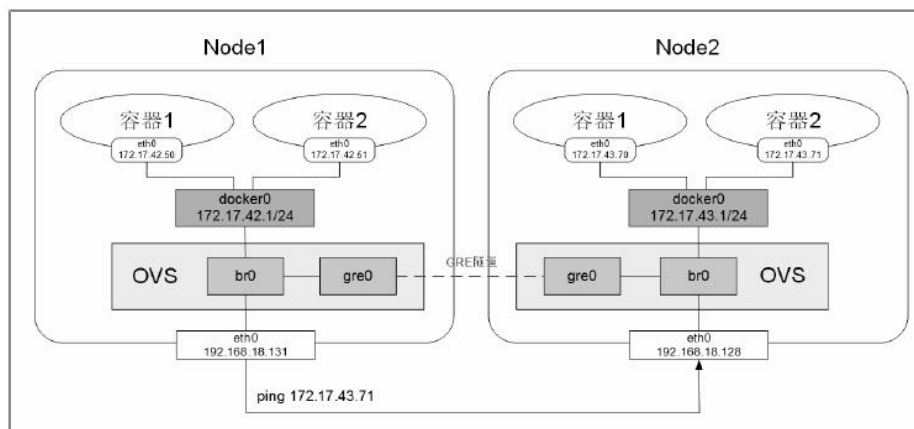


图2.2 通过Open vSwitch打通网络

首先，确保节点192.168.18.128的Docker0采用172.17.43.0/24网段，而192.168.18.131的Docker0采用172.17.42.0/24网段，对应参数为docker daemon的启动参数--bip设置的值。

Open vSwitch的安装和配置方法如下。

1) 在两个Node上安装ovs

```
# yum install openvswitch-2.4.0-1.x86_64.rpm
```

禁止selinux，配置后重启Linux：

```
# vi /etc/selinux/config
SELINUX=disabled
```

查看Open vSwitch的服务状态，应该启动两个进程：ovsdb-server与ovs-vswitchd。

```
# service openvswitch status  
  
ovsdb-server is running with pid 2429  
ovs-vswitchd is running with pid 2439
```

查看Open vSwitch的相关日志，确认没有异常：

```
# more /var/log/messages |grep openv  
  
Nov  2 03:12:52 docker128 openvswitch: Starting ovsdb-  
server [ OK ]  
  
Nov  2 03:12:52 docker128 openvswitch: Configuring  
Open vSwitch system IDs [ OK ]  
  
Nov  2 03:12:52 docker128 kernel: openvswitch: Open  
vSwitch switching datapath  
  
Nov  2 03:12:52 docker128 openvswitch: Inserting  
openvswitch module [ OK ]
```

注意上述操作需要在两个节点机器上分别执行完成。

2) 创建网桥和GRE隧道

接下来需要在每个Node上建立ovs的网桥br0，然后在网桥上创建一个GRE隧道连接对端网桥，最后把ovs的网桥br0作为一个端口连接到docker0这个Linux网桥上（可以认为是交换机互联），这样一来，两个节点机器上的docker0网段就能互通了。

下面以节点机器192.168.18.131为例，具体的操作步骤如下。

(1) 创建ovs网桥:

```
# ovs-vsctl add-br br0
```

(2) 创建GRE隧道连接对端，remote_ip为对端eth0的网卡地址：

```
# ovs-vsctl add-port br0 gre1 -- set interface gre1  
type=gre option:remote_ip=192.168.18.128
```

(3) 添加br0到本地docker0，使得容器流量通过OVS流经tunnel：

```
# brctl addif docker0 br0
```

(4) 启动br0与docker0网桥：

```
# ip link set dev br0 up  
# ip link set dev docker0 up
```

(5) 添加路由规则。由于192.168.18.128与192.168.18.131的docker0网段分别为172.17.43.0/24与172.17.42.0/24，这两个网段的路由都需要经过本机的docker0网桥路由，其中一个24网段是通过OVS的GRE隧道到达对端的，因此需要在每个Node上添加通过docker0网桥转发的172.17.0.0/16段的路由规则：

```
# ip route add 172.17.0.0/16 dev docker0
```

(6) 清空Docker自带的Iptables规则及Linux的规则，后者存在拒绝icmp报文通过防火墙的规则：

```
# iptables -t nat -F; iptables -F
```

在192.168.18.131上完成上述步骤后，在192.168.18.128节点执行同样的操作，注意，GRE隧道里的IP地址要改为对端节点（192.168.18.131）的IP地址。

配置完成后，192.168.18.131的IP地址、docker0的IP地址及路由等重要信息显示如下：

```
# ip addr

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd
00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:55:5e:c3 brd ff:ff:ff:ff:ff:ff
    inet 192.168.18.131/24 brd 192.168.18.255 scope
global dynamic eth0
        valid_lft 1369sec preferred_lft 1369sec
3: ovs-system: <BROADCAST,MULTICAST> mtu 1500 qdisc
noop state DOWN
    link/ether a6:15:c3:25:cf:33 brd ff:ff:ff:ff:ff:ff
4: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
qdisc noqueue master docker0 state UNKNOWN
```



```
link/ether 92:8d:d0:a4:ca:45 brd ff:ff:ff:ff:ff:ff
5: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
qdisc noqueue state UP
link/ether 02:42:44:8d:62:11 brd ff:ff:ff:ff:ff:ff
inet 172.17.42.1/24 scope global docker0
valid_lft forever preferred_lft forever
```

同样，192.168.18.128节点的重要信息如下：

```
# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
state UNKNOWN
link/loopback 00:00:00:00:00:00 brd
00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
qdisc pfifo_fast state UP qlen 1000
link/ether 00:0c:29:e8:02:c7 brd ff:ff:ff:ff:ff:ff
inet 192.168.18.128/24 brd 192.168.18.255 scope
global dynamic eth0
valid_lft 1356sec preferred_lft 1356sec
3: ovs-system: <BROADCAST,MULTICAST> mtu 1500 qdisc
noop state DOWN
link/ether fa:6c:89:a2:f2:01 brd ff:ff:ff:ff:ff:ff
4: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
qdisc noqueue master docker0 state UNKNOWN
```

```
link/ether ba:89:14:e0:7f:43 brd ff:ff:ff:ff:ff:ff
5: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
qdisc noqueue state UP
    link/ether 02:42:63:a8:14:d5 brd ff:ff:ff:ff:ff:ff
    inet 172.17.43.1/24 scope global docker0
        valid_lft forever preferred_lft forever
```

3) 两个Node上容器之间的互通测试

首先，在192.168.18.128节点上ping192.168.18.131上的docker0地址：172.17.42.1，验证网络互通性：

```
# ping 172.17.42.1
PING 172.17.42.1 (172.17.42.1) 56(84) bytes of data.
 64 bytes from 172.17.42.1: icmp_seq=1 ttl=64 time=1.57
ms
    64 bytes from 172.17.42.1: icmp_seq=2 ttl=64
time=0.966 ms
    64 bytes from 172.17.42.1: icmp_seq=3 ttl=64 time=1.01
ms
    64 bytes from 172.17.42.1: icmp_seq=4 ttl=64 time=1.00
ms
    64 bytes from 172.17.42.1: icmp_seq=5 ttl=64 time=1.22
ms
    64 bytes from 172.17.42.1: icmp_seq=6 ttl=64
time=0.996 ms
```

下面我们通过 **tshark** 抓包工具来分析流量走向。首先，在 192.168.18.128 节点上监听 **br0** 上是否有 **GRE** 报文，执行下面的命令，我们发现 **br0** 上并没有 **GRE** 报文：

```
# tshark -i br0 -R ip proto GRE
tshark: -R without -2 is deprecated. For single-pass
filtering use -Y.
Running as user "root" and group "root". This could be
dangerous.
Capturing on 'br0'
^C
```

而在 **eth0** 上抓包，则发现了 **GRE** 封装的 **ping** 包报文通过，说明 **GRE** 是在承载网的物理网上完成的封包过程：

```
# tshark -i eth0 -R ip proto GRE
tshark: -R without -2 is deprecated. For single-pass
filtering use -Y.
Running as user "root" and group "root". This could be
dangerous.
Capturing on 'eth0'
1    0.000000  172.17.43.1 -> 172.17.42.1  ICMP 136
Echo (ping) request  id=0x0970, seq=180/46080, ttl=64
2    0.000892  172.17.42.1 -> 172.17.43.1  ICMP 136
Echo (ping) reply    id=0x0970, seq=180/46080, ttl=64
(request in 1)
2    3    1.002014  172.17.43.1 -> 172.17.42.1  ICMP 136
```

```
Echo (ping) request  id=0x0970, seq=181/46336, ttl=64
      4    1.002916  172.17.42.1 -> 172.17.43.1  ICMP 136
Echo (ping) reply      id=0x0970, seq=181/46336, ttl=64
(request in 3)
      4    5    2.004101  172.17.43.1 -> 172.17.42.1  ICMP 136
Echo (ping) request  id=0x0970, seq=182/46592, ttl=64
```

至此，基于OVS的网络搭建成功，由于GRE是点对点隧道通信方式，所以如果有多个Node，则需要建立 $N \times (N-1)$ 条GRE隧道，即所有Node组成一个网状的网络，实现全网互通。

3.直接路由

通过在每个Node上添加到其他Node上docker0的静态路由规则，就可以将不同物理机的docker0网桥互联互通。图2.3描述了在两个Node之间打通网络的情况。

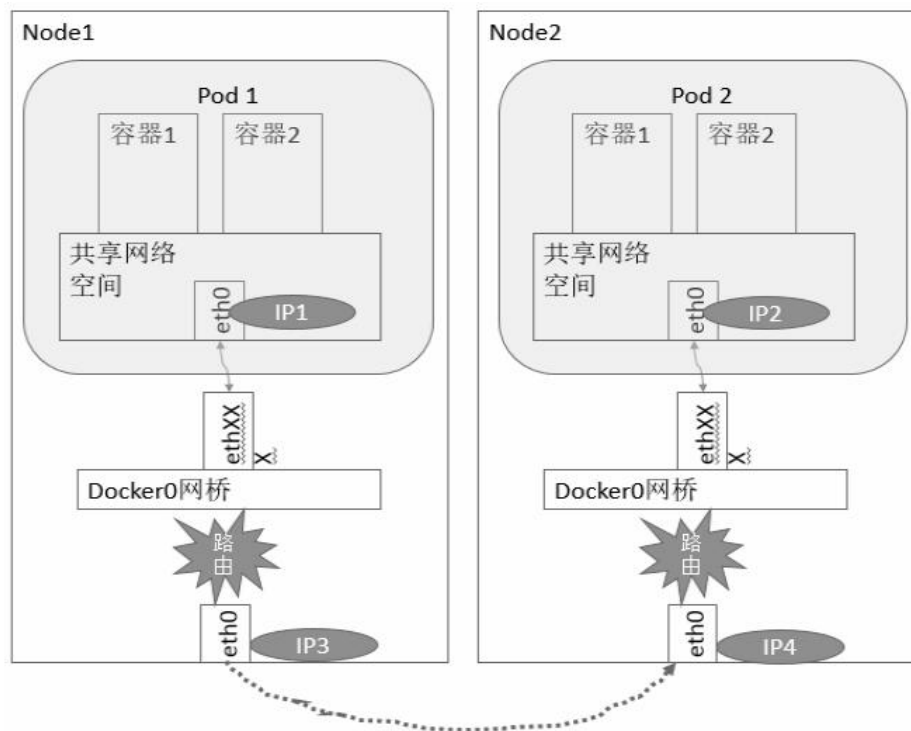


图2.3 以直接路由方式实现Pod到Pod的通信

使用这种方案，只需要在每个Node的路由表中增加到对方docker0的静态路由转发规则。

例如Pod1所在docker0网桥的IP子网是10.1.10.0，Node地址为192.168.1.128；而Pod2所在docker0网桥的IP子网是10.1.20.0，Node地址为192.168.1.129。

在Node1上用route add命令增加一条到Node2上docker0的静态路由规则：

```
route add -net 10.1.20.0 netmask 255.255.255.0 gw
192.168.1.129
```

同样，在Node2上增加一条到Node1上docker0的静态路由规则：

```
route add -net 10.1.10.0 netmask 255.255.255.0 gw
192.168.1.128
```

在Node1上通过ping命令验证到Node2上docker0的网络连通性。这里10.1.20.1为Node2上docker0网桥自身的IP地址。

```
$ ping 10.1.20.1
PING10.1.20.1 (10.1.20.1) 56(84) bytes of data.
 64 bytes from 10.1.20.1: icmp_seq=1 ttl=62 time=1.15
ms
 64 bytes from 10.1.20.1: icmp_seq=2 ttl=62 time=1.16
ms
 64 bytes from 10.1.20.1: icmp_seq=3 ttl=62 time=1.57
ms
.....
```

可以看到，路由转发规则生效，Node1可以直接访问到Node2上的docker0网桥，进一步也可以访问到属于docker0网段的容器应用了。

不过，集群中机器的数量通常可能很多。假设有100台服务器，那么就需要在每台服务器上手工添加到另外99台服务器docker0的路由规则。为了减少手工操作，可以使用Quagga软件来实现路由规则的动态添加。Quagga软件的主页为<http://www.quagga.net>。

除了在这每台服务器上安装Quagga软件并启动，还可以使用Quagga容器来运行（例如index.alauda.cn/georce/router）。在这每台Node上下载

该Docker镜像:

```
$ docker pull index.alauda.cn/georce/router
```

在运行Quagga容器之前，需要确保每个Node上docker0网桥的子网地址不能重叠，也不能与物理机所在的网络重叠，这需要网络管理员的仔细规划。

下面以3个Node为例，每台Node的docker0网桥的地址如下（前提是Node物理机的IP地址不是10.1.X.X地址段）：

```
Node 1: # ifconfig docker0 10.1.10.1/24
Node 2: # ifconfig docker0 10.1.20.1/24
Node 3: # ifconfig docker0 10.1.30.1/24
```

然后在每个Node上启动Quagga容器。需要说明的是，Quagga需要以--privileged特权模式运行，并且指定--net=host，表示直接使用物理机的网络：

```
$ docker run -itd --name=router --privileged --
net=host index.alauda.cn/georce/router
```

启动成功后，Quagga会相互学习来完成到其他机器的docker0路由规则的添加。

一段时间后，在Node1上使用route-n命令来查看路由表，可以看到Quagga自动添加了两条到Node2和到Node3上docker0的路由规则。

# route -n						
Kernel IP routing table						
	Destination			Gateway		Genmask
Flags	Metric	Ref	Use	Iface		
	0.0.0.0		192.168.1.128	0.0.0.0		UG
0	0	0 eth0				
	10.1.10.0		0.0.0.0	255.255.255.0		U
0	0	0 docker0				
	10.1.20.0		192.168.1.129	255.255.255.0		UG
20	0	0 eth0				
	10.1.30.0		192.168.1.130	255.255.255.0		UG
20	0	0 eth0				

在Node2上查看路由表，可以看到自动添加了两条到Node1和Node3上docker0的路由规则。

# route -n						
Kernel IP routing table						
	Destination		Gateway		Genmask	Flags
Metric	Ref	Use	Iface			
	0.0.0.0		192.168.1.129	0.0.0.0		UG
0	0	0 eth0				
	10.1.20.0		0.0.0.0	255.255.255.0		U
0	0	0 docker0				
	10.1.10.0		192.168.1.128	255.255.255.0		UG
20	0	0 eth0				

	10.1.30.0	192.168.1.130	255.255.255.0	UG
20	0	0	eth0	

至此，所有Node上的docker0都可以互联互通了。

2.2 kubectl命令行工具用法详解

kubectl作为客户端CLI工具，可以让用户通过命令行的方式对Kubernetes集群进行操作。本节对kubectl的子命令和用法进行详细说明。

2.2.1 kubectl用法概述

kubectl命令行的语法如下：

```
$ kubectl [command] [TYPE] [NAME] [flags]
```

其中，`command`、`TYPE`、`NAME`、`flags`的含义如下。

(1) `command`：子命令，用于操作Kubernetes集群资源对象的命令，例如`create`、`delete`、`describe`、`get`、`apply`等。

(2) `TYPE`：资源对象的类型，区分大小写，能以单数形式、复数形式或者简写形式表示。例如以下3种`TYPE`是等价的。

```
$ kubectl get pod pod1
$ kubectl get pods pod1
$ kubectl get po pod1
```

(3) `NAME`：资源对象的名称，区分大小写。如果不指定名称，则系统将返回属于`TYPE`的全部对象的列表，例如`$kubectl get pods`将返回所有Pod的列表。

(4) `flags`：kubectl子命令的可选参数，例如使用“-s”指定apiserver的URL地址而不用默认值。

kubectl可操作的资源对象类型如表2.9所示。

表2.9 kubectl可操作的资源对象类型

资源对象的名称	缩 写
componentstatuses	cs
daemonsets	ds
deployments	
events	ev
endpoints	ep
horizontalpodautoscalers	hpa
ingresses	ing
jobs	
limitranges	limits
nodes	no
namespaces	ns
pods	po
persistentvolumes	pv
persistentvolumeclaims	pvc
resourcequotas	quota
replicationcontrollers	rc
secrets	
serviceaccounts	
services	svc

在一个命令行中也可以同时对多个资源对象进行操作，以多个TYPE和NAME的组合表示，示例如下。

- 获取多个Pod的信息：

```
$ kubectl get pods pod1 pod2
```

- 获取多种对象的信息：

```
$ kubectl get pod/pod1 rc/rc1
```

- 同时应用多个yaml文件，以多个-f file参数表示：

```
$ kubectl get pod -f pod1.yaml -f pod2.yaml  
$ kubectl create -f pod1.yaml -f rc1.yaml -f  
service1.yaml
```

2.2.2 kubectl子命令详解

kubectl的子命令非常丰富，涵盖了对Kubernetes集群的主要操作，包括资源对象的创建、删除、查看、修改、配置、运行等。详细的子命令如表2.10所示。

表2.10 kubectl子命令详解

子 命 令	语 法	说 明
annotate	kubectl annotate [--overwrite] (-f FILENAME TYPE NAME) KEY_1=VAL_1 ... KEY_N=VAL_N [--resource-version=version] [flags]	添加或更新资源对象的 annotation 信息
api-versions	kubectl api-versions [flags]	列出当前系统支持的 API 版本列表，格式为 “group/version”
apply	kubectl apply -f FILENAME [flags]	从配置文件或 stdin 中对资源对象进行配置更新
attach	kubectl attach POD -c CONTAINER [flags]	附着到一个正在运行的容器上
autoscale	kubectl autoscale (-f FILENAME TYPE NAME TYPE/NAME) [--min=MINPODS] --max=MAXPODS [--cpu-percent=CPU] [flags]	对 Deployment、ReplicaSet 或 ReplicationController 进行水平自动扩缩容的设置
cluster-info	kubectl cluster-info [flags] kubectl cluster-info [command]	显示集群信息
completion	kubectl completion SHELL [flags]	输出 shell 命令的执行结果码 (bash 或 zsh)
config	kubectl config SUBCOMMAND [flags] kubectl config [command]	修改 kubeconfig 文件
convert	kubectl convert -f FILENAME [flags]	转换配置文件为不同的 API 版本
cordon	kubectl cordon NODE [flags]	将 Node 标记为 unschedulable，即“隔离”出集群调度范围
create	kubectl create -f FILENAME [flags] kubectl create [command]	从配置文件或 stdin 中创建资源对象
delete	kubectl delete ([-f FILENAME] TYPE [(NAME -l label --all)]) [flags]	根据配置文件、stdin、资源名称或 label selector 删除资源对象
describe	kubectl describe (-f FILENAME TYPE [NAME_PREFIX /NAME -l label]) [flags]	描述一个或多个资源对象的详细信息
drain	kubectl drain NODE [flags]	首先将 Node 设置为 unschedulable，然后删除该 Node 上运行的所有 Pod，但不会删除不由 apiserver 管理的 Pod
edit	kubectl edit (RESOURCE/NAME -f FILENAME) [flags]	编辑资源对象的属性，在线更新

续表

子 命 令	语 法	说 明
exec	kubectl exec POD [-c CONTAINER] -- COMMAND [args...] [flags]	执行一个容器中的命令
explain	kubectl explain RESOURCE [flags]	对资源对象属性的详细说明
expose	kubectl expose (-f FILENAME TYPE NAME) [--port=port] [--protocol=TCP UDP] [--target-port=number-or-name] [--name=name] [--external-ip=external-ip-of-service] [--type=type] [flags]	将已经存在的一个 RC、Service、Deployment 或 Pod 暴露为一个新的 Service
get	kubectl get [(--o --output=json yaml wide go-template=... go-template-file=... jsonpath=... jsonpath-file=...)] (TYPE [NAME -l label] TYPE/NAME ...) [flags]	显示一个或多个资源对象的概要信息
label	kubectl label [--overwrite] (-f FILENAME TYPE NAME) KEY_1=VAL_1 ... KEY_N=VAL_N [--resource-version=version] [flags]	设置或更新资源对象的 labels
logs	kubectl logs [-f] [-p] POD [-c CONTAINER] [flags]	屏幕打印一个容器的日志
namespace	kubectl namespace [namespace] [flags]	已被 kubectl config set-context 替代
patch	kubectl patch (-f FILENAME TYPE NAME) -p PATCH [flags]	以 merge 形式对资源对象的部分字段的值进行修改
port-forward	kubectl port-forward POD [LOCAL_PORT:REMOTE_PORT [...[LOCAL_PORT_N:REMOTE_PORT_N]]] [flags]	将本机的某个端口号映射到 Pod 的端口号，通常用于测试工作
proxy	kubectl proxy [--port=PORT] [--www=static-dir] [--www-prefix=prefix] [--api-prefix=prefix] [flags]	将本机某个端口号映射到 apiserver
replace	kubectl replace -f FILENAME [flags]	从配置文件或 stdin 替换资源对象
rolling-update	kubectl rolling-update OLD_CONTROLLER_NAME (NEW_CONTROLLER_NAME) --image=NEW_CONTAINER_IMAGE -f NEW_CONTROLLER_SPEC [flags]	对 RC 进行滚动升级
rollout	kubectl rollout SUBCOMMAND [flags] kubectl rollout [command]	对 Deployment 进行管理，可用操作包括：history、pause、resume、undo、status
run	kubectl run NAME --image=image [--env="key=value"] [--port=port] [--replicas=replicas] [--dry-run=bool] [--overrides=inline-json] [--command] -- [COMMAND] [args...] [flags]	基于一个镜像在 Kubernetes 集群上启动一个 Deployment
scale	kubectl scale [--resource-version=version] [--current-replicas=count] --replicas=COUNT (-f FILENAME TYPE NAME) [flags]	扩容、缩容一个 Deployment、ReplicaSet、RC 或 Job 中 Pod 的数量
set	kubectl set SUBCOMMAND [flags] kubectl set [command]	设置资源对象的某个特定信息，目前仅支持修改容器的镜像
taint	kubectl taint NODE NAME KEY_1=VAL_1:TAINT_EFFECT_1 ... KEY_N=VAL_N:TAINT_EFFECT_N [flags]	设置 Node 的 taint 信息，用于将特定的 Pod 调度到特定的 Node 的操作，为 Alpha 版本功能
uncordon	kubectl uncordon NODE [flags]	将 Node 设置为 schedulable
version	kubectl version [flags]	打印系统的版本信息

2.2.3 kubectl参数列表

kubectl命令行的公共启动参数如表2.11所示。

表2.11 kubectl命令行公共参数

参数名和取值示例	说 明
--alsologtostderr[=false]	设置为 true 表示将日志输出到文件的同时输出到 stderr
--as=""	设置本次操作的用户名
--certificate-authority=""	用于 CA 授权的 cert 文件路径
--client-certificate=""	用于 TLS 的客户端证书文件路径
--client-key=""	用于 TLS 的客户端 key 文件路径
--cluster=""	设置要使用的 kubeconfig 中的 cluster 名
--context=""	设置要使用的 kubeconfig 中的 context 名
--insecure-skip-tls-verify[=false]	设置为 true 表示跳过 TLS 安全验证模式，将使得 HTTPS 连接不安全
--kubeconfig=""	kubeconfig 配置文件路径，在配置文件中包括 Master 地址信息及必要的认证信息
--log-backtrace-at=:0	记录日志每到“file:行号”时打印一次 stack trace
--log-dir=""	日志文件路径
--log-flush-frequency=5s	设置 flush 日志文件的时间间隔
--logtostderr[=true]	设置为 true 表示将日志输出到 stderr，不输出到日志文件
--match-server-version[=false]	设置为 true 表示客户端版本号需要与服务端一致
--namespace=""	设置本次操作所在的 namespace
--password=""	设置 apiserver 的 basic authentication 的密码
-s, --server=""	设置 apiserver 的 URL 地址，默认为 localhost:8080
--stderrthreshold=2	在该 threshold 级别之上的日志将输出到 stderr
--token=""	设置访问 apiserver 的安全 token
--user=""	指定 kubeconfig 用户名
--username=""	设置 apiserver 的 basic authentication 的用户名
--v=0	glog 日志级别
--vmodule=	glog 基于模块的详细日志级别

每个子命令（如create、delete、get等）还有特定的flags参数，可以通过\$kubectl[command]--help命令进行查看。

2.2.4 kubectl输出格式

kubectl命令可以用多种格式对结果进行显示，输出的格式通过-o参数指定：

```
$ kubectl [command] [TYPE] [NAME] -o=<output_format>
```

根据不同子命令的输出结果，可选的输出格式如表2.12所示。

表2.12 kubectl命令输出格式列表

输出格式	说 明
-o=custom-columns=<spec>	根据自定义列名进行输出，以逗号分隔
-o=custom-columns-file=<filename>	从文件中获取自定义列名进行输出
-o=json	以 JSON 格式显示结果
-o=jsonpath=<template>	输出 jsonpath 表达式定义的字段信息
-o=jsonpath-file=<filename>	输出 jsonpath 表达式定义的字段信息，来源于文件
-o=name	仅输出资源对象的名称
-o=wide	输出额外信息。对于 Pod，将输出 Pod 所在的 Node 名
-o=yaml	以 yaml 格式显示结果

常用的输出格式示例如下。

(1) 显示Pod的更多信息：

```
$ kubectl get pod <pod-name> -o wide
```

(2) 以yaml格式显示Pod的详细信息：

```
$ kubectl get pod <pod-name> -o yaml
```

(3) 以自定义列名显示Pod的信息:

```
$ kubectl get pod <pod-name> -o=custom-  
columns=NAME:.metadata.name,RSRC:.metadata.resourceVersion
```

(4) 基于文件的自定义列名输出:

```
$ kubectl get pods <pod-name> -o=custom-columns-  
file=template.txt
```

template.txt文件的内容为:

NAME	RSRC
metadata.name	metadata.resourceVersion

输出结果为:

NAME	RSRC
pod-name	52305

另外, 还可以将输出结果按某个字段排序, 通过--sort-by参数以jsonpath表达式进行指定:

```
$ kubectl [command] [TYPE] [NAME] --sort-by=  
<jsonpath_exp>
```

例如, 按照名字进行排序:

```
$ kubectl get pods --sort-by=.metadata.name
```

2.2.5 kubectl操作示例

本节对一些常用的kubectl操作进行示例。

1.创建资源对象

根据yaml配置文件一次性创建service和rc:

```
$ kubectl create -f my-service.yaml -f my-rc.yaml
```

根据<directory>目录下所有.yaml、.yml、.json文件的定义进行创建操作:

```
$ kubectl create -f <directory>
```

2.查看资源对象

查看所有Pod列表:

```
$ kubectl get pods
```

查看rc和service列表:

```
$ kubectl get rc,service
```

3.描述资源对象

显示Node的详细信息:

```
$ kubectl describe nodes <node-name>
```

显示Pod的详细信息:

```
$ kubectl describe pods/<pod-name>
```

显示由RC管理的Pod的信息:

```
$ kubectl describe pods <rc-name>
```

4.删除资源对象

基于pod.yaml定义的名称删除Pod:

```
$ kubectl delete -f pod.yaml
```

删除所有包含某个label的Pod和service:

```
$ kubectl delete pods,services -l name=<label-name>
```

删除所有Pod:

```
$ kubectl delete pods --all
```

5.执行容器的命令

执行Pod的date命令，默认使用Pod中的第1个容器执行：

```
$ kubectl exec <pod-name> date
```

指定Pod中某个容器执行date命令：

```
$ kubectl exec <pod-name> -c <container-name> date
```

通过bash获得Pod中某个容器的TTY，相当于登录容器：

```
$ kubectl exec -ti <pod-name> -c <container-name> /bin/bash
```

6.查看容器的日志

查看容器输出到stdout的日志：

```
$ kubectl logs <pod-name>
```

跟踪查看容器的日志，相当于tail -f命令的结果：

```
$ kubectl logs -f <pod-name> -c <container-name>
```

2.3 Guestbook示例：Hello World

在对Kubernetes的容器应用进行详细说明之前，让我们先通过一个由3个微服务组成的留言板（Guestbook）系统的搭建，对Kubernetes对容器应用的基本操作和用法进行初步介绍。本章后面的章节将基于该案例和其他示例，进一步深入Pod、RC、Service等核心对象的用法和技巧，对Kubernetes的应用管理进行全面讲解。

Guestbook留言板系统将通过Pod、RC、Service等资源对象搭建完成，成功启动后在网页中显示一条“Hello World”留言。其系统架构是一个基于PHP+Redis的分布式Web应用，前端PHP Web网站通过访问后端的Redis来完成用户留言的查询和添加等功能。同时Redis以Master+Slave的模式进行部署，实现数据的读写分离能力。

留言板系统的部署架构如图2.4所示。Web层是一个基于PHP页面的Apache服务，启动3个实例组成集群，为客户端（例如浏览器）对网站的访问提供负载均衡。Redis Master启动1个实例用于写操作（添加留言），RedisSlave启动两个实例用于读操作（读取留言）。RedisMaster与Slave的数据同步由Redis具备的数据同步机制完成。

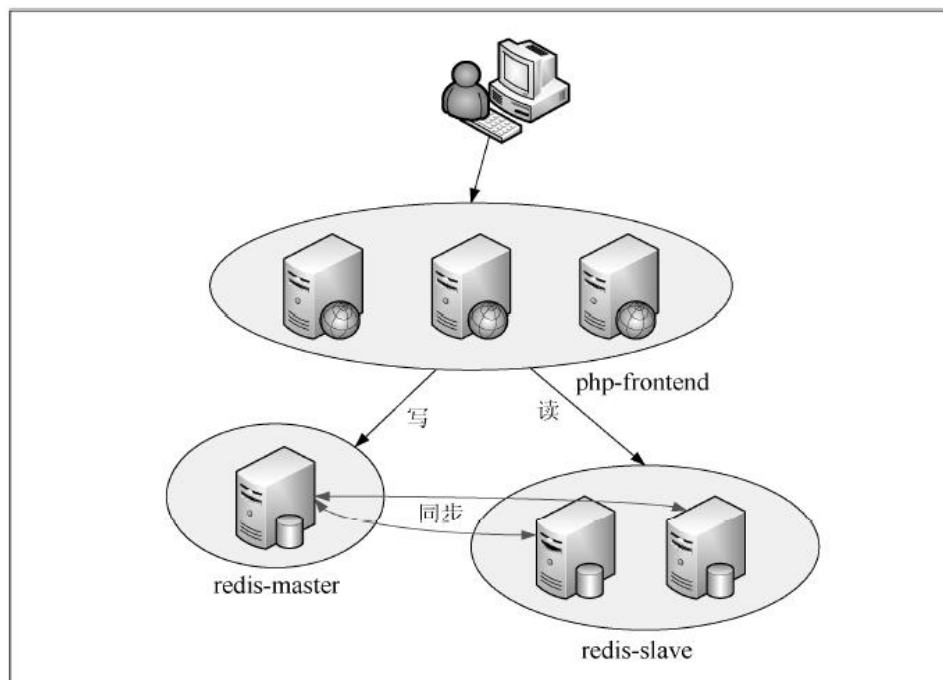


图2.4 留言板的系统部署架构图

在本例中将要用到 3 个 Docker 镜像，下载地址为 <https://hub.docker.com/u/kubeguide/>。

- **redis-master**: 用于前端Web应用进行“写”留言操作的Redis服务，其中已经保存了一条内容为“Hello World!”的留言。
- **guestbook-redis-slave**: 用于前端Web应用进行“读”留言操作的Redis服务，并与Redis-Master的数据保持同步。
- **guestbook-php-frontend**: PHP Web服务，在网页上展示留言的内容，也提供一个文本输入框供访客添加留言。

如图2.5所示为Hello World案例所采用的Kubernetes部署架构，这里Master与Node的服务处于同一个虚拟机中。通过创建redis-master服务、redis-slave服务和php-frontend服务来实现整个系统的搭建。

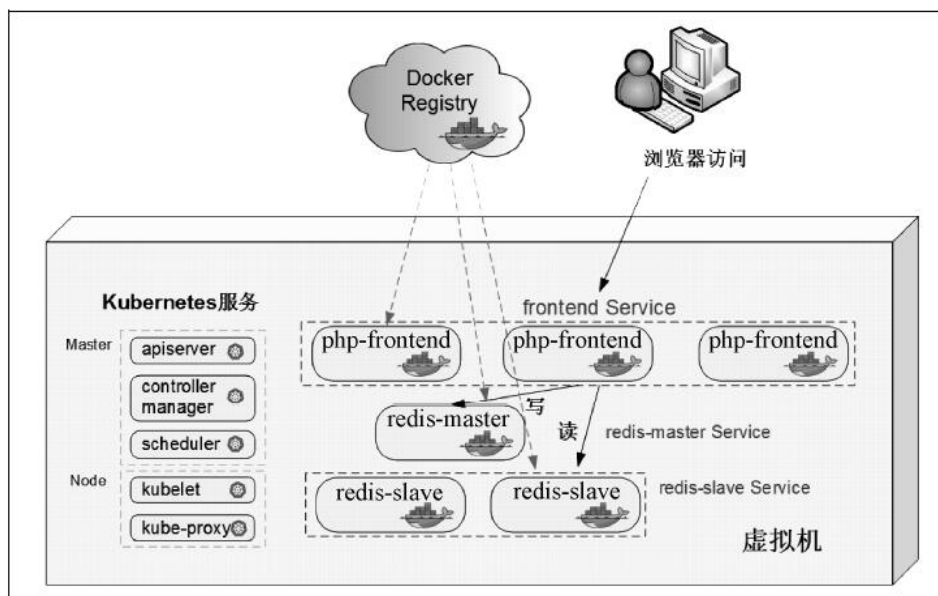


图2.5 Kubernetes部署架构图

2.3.1 创建redis-master RC和Service

我们可以先定义Service，然后定义一个RC来创建和控制相关联的Pod，或者先定义RC来创建Pod，然后定义与之关联的Service，这里我们采用后一种方法。

首先为redis-master创建一个名为redis-master的RC定义文件redis-master-controller.yaml。yaml的语法类似于PHP的语法，对于空格的个数有严格的要求，详见<http://yaml.org>。

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-master
  labels:
    name: redis-master
spec:
  replicas: 1
  selector:
    name: redis-master
  template:
    metadata:
      labels:
        name: redis-master
    spec:
```

```
containers:
  - name: master
    image: kubeguide/redis-master
    ports:
      - containerPort: 6379
```

其中，`kind`字段的值为“`ReplicationController`”，表示这是一个RC；`spec.selector`是RC的Pod选择器，即监控和管理拥有这些标签（Label）的Pod实例，确保当前集群上始终有且仅有`replicas`个Pod实例在运行，这里我们设置`replicas=1`表示只运行一个（名为`redis-master`的）Pod实例，当集群中运行的Pod数量小于`replicas`时，RC会根据`spec.template`段定义的Pod模板来生成一个新的Pod实例，`labels`属性指定了该Pod的标签，注意，这里的`labels`必须匹配RC的`spec.selector`，否则此RC就会陷入“只为他人做嫁衣”的悲惨世界中，永无翻身之时。

创建好`redis-master-controller.yaml`文件以后，我们在Master节点执行命令：`kubectl create -f <config_file>`，将它发布到Kubernetes集群中，就完成了`redis-master`的创建过程：

```
$ kubectl create -f redis-master-controller.yaml
replicationcontroller "redis-master" created
```

系统提示“`redis-master`”表示创建成功。然后我们用`kubectl`命令查看刚刚创建的`redis-master`：

```
$ kubectl get rc
```

NAME	DESIRED	CURRENT	AGE
------	---------	---------	-----

redis-master	1	1	5m
--------------	---	---	----

接下来运行 `kubectl get pods` 命令来查看当前系统中的Pod列表信息，我们看到一个名为 `redis-master-xxxxx` 的Pod实例，这是Kubernetes根据 `redis-master` 这个RC的定义自动创建的Pod。RC会给每个Pod实例在用户设置的 `name` 后补充一段UUID，以区分不同的实例。由于Pod的调度和创建需要花费一定的时间，比如需要一定的时间来确定调度到哪个节点上，以及下载Pod的相关镜像，所以一开始我们看到Pod的状态将显示为 `Pending`。当Pod成功创建完成以后，状态会被更新为 `Running`。如果Pod一直处于 `Pending` 状态，则请参看第5章的查错说明。

\$ kubectl get pods			
NAME		READY	STATUS
RESTARTS	AGE		
	redis-master-b03io	1/1	Running
0	1h		

`redis-master` Pod已经创建并正常运行了，接下来我们就创建一个与之关联的 `Service`（服务）定义文件（文件名为 `redis-master-service.yaml`），完整的内容如下：

apiVersion: v1
kind: Service
metadata:
name: redis-master
labels:

```
        name: redis-master
spec:
  ports:
    - port: 6379
      targetPort: 6379
  selector:
    name: redis-master
```

其中 `metadata.name` 是 Service 的服务名（`ServiceName`），`spec.selector` 确定了选择哪些 Pod，本例中的定义表明将选择设置过 `name=redis-master` 标签的 Pod。`port` 属性定义的是 Service 的虚拟端口号，`targetPort` 属性指定后端 Pod 容器应用监听的端口号。

运行 `kubectl create` 命令创建该 service:

```
$ kubectl create -f redis-master-service.yaml
service "redis-master" created
```

系统提示“service”redis-master“created”表示创建成功。然后运行 `kubectl get` 命令可以查看到刚刚创建的 service:

```
$ kubectl get services
```

	NAME	CLUSTER-IP	EXTERNAL-IP
PORT(S)	AGE		
	redis-master	169.169.208.57	<none>
6379/TCP	13m		

注意到redis-master服务被分配了一个值为169.169.208.57的虚拟IP地址，随后，Kubernetes集群中其他新创建的Pod就可以通过这个虚拟IP地址+端口6379来访问这个服务了。在本例中将要创建的redis-slave和frontend两组Pod都将通过169.169.208.57: 6379来访问redis-master服务。

但由于IP地址是在服务创建后由Kubernetes系统自动分配的，在其他Pod中无法预先知道某个Service的虚拟IP地址，因此需要一个机制来找到这个服务。为此，Kubernetes巧妙地使用了Linux环境变量（Environment Variable），在每个Pod的容器里都增加了一组Service相关的环境变量，用来记录从服务名到虚拟IP地址的映射关系。以redis-master服务为例，在容器的环境变量中会增加下面两条记录：

```
REDIS_MASTER_SERVICE_HOST=169.169.144.74  
REDIS_MASTER_SERVICE_PORT=6379
```

于是，redis-slave和frontend等Pod中的应用程序就可以通过环境变量REDIS_MASTER_SERVICE_HOST得到redis-master服务的虚拟IP地址，通过环境变量REDIS_MASTER_SERVICE_PORT得到redis-master服务的端口号，这样就完成了对服务地址的查询功能。

2.3.2 创建redis-slave RC和Service

现在我们已经成功启动了redis-master服务，接下来我们继续完成redis-slave服务的创建过程。在本案例中会启动redis-slave服务的两个副本，每个副本上的Redis进程都与redis-master进行数据同步，与redis-master共同组成了一个具备读写分离能力的Redis集群。留言板的PHP网页将通过访问redis-slave服务来读取留言数据。与之前的redis-master服务的创建过程一样，首先创建一个名为redis-slave的RC定义文件redis-slave-controller.yaml，完整内容如下：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-slave
  labels:
    name: redis-slave
spec:
  replicas: 2
  selector:
    name: redis-slave
  template:
    metadata:
      labels:
        name: redis-slave
```

```
spec:
  containers:
    - name: slave
      image: kubeguide/guestbook-redis-slave
      env:
        - name: GET_HOSTS_FROM
          value: env
      ports:
        - containerPort: 6379
```

在容器的配置部分设置了一个环境变量GET_HOSTS_FROM=env，意思是从环境变量中获取redis-master服务的IP地址信息。

redis-slave镜像中的启动脚本/run.sh的内容为:

```
if [[ ${GET_HOSTS_FROM:-dns} == "env" ]]; then
    redis-server --slaveof ${REDIS_MASTER_SERVICE_HOST}
6379
else
    redis-server --slaveof redis-master 6379
fi
```

在创建redis-slave Pod时，系统将自动在容器内部生成之前已经创建好的redis-master service相关的环境变量，所以redis-slave应用程序redis-server可以直接使用环境变量REDIS_MASTER_SERVICE_HOST来获取redis-master服务的IP地址。

如果在容器配置部分不设置该env，则将使用redis-master服务的名称“redis-master”来访问它，这将使用DNS方式的服务发现，需要预先启动Kubernetes集群的skydns服务，详见2.5.4节的说明。

运行kubectl create命令：

```
$ kubectl create -f redis-slave-controller.yaml
Replicationcontrollers "redis-slave" created
```

运行kubectl get命令查看RC：

```
$ kubectl get rc
```

NAME	DESIRED	CURRENT	AGE
redis-master	1	1	1h
redis-slave	2	2	1h

查看RC创建的Pod，可以看到有两个redis-slave Pod在运行：

```
$ kubectl get pods
```

NAME	READY	STATUS
redis-master-b03io	1/1	Running
redis-slave-10ahl	1/1	Running
redis-slave-c5y10	1/1	Running

然后创建redis-slave服务。类似于redis-master服务，与redis-slave相关的一组环境变量也将在后续新建的frontend Pod中由系统自动生成。

配置文件redis-slave-service.yaml的内容如下：

```
apiVersion: v1
kind: Service
metadata:
  name: redis-slave
  labels:
    name: redis-slave
spec:
  ports:
    - port: 6379
  selector:
    name: redis-slave
```

运行kubectl创建Service:

```
$ kubectl create -f redis-slave-service.yaml
services/redis-slave
```

通过kubectl查看创建的Service:

```
$ kubectl get services
```

NAME	CLUSTER-IP	EXTERNAL-IP
PORT(S)	AGE	

	frontend	169.169.167.153	<nodes>
80/TCP	25m		
	redis-master	169.169.208.57<none>	6379/TCP
25m			
	redis-slave	169.169.78.102<none>	6379/TCP
25m			

2.3.3 创建frontend RC和Service

类似地，定义frontend的RC配置文件——frontend-controller.yaml，内容如下：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  replicas: 3
  selector:
    name: frontend
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
        - name: frontend
          image: kubeguide/guestbook-php-frontend
          env:
```

```
- name: GET_HOSTS_FROM
  value: env
ports:
- containerPort: 80
```

在容器的配置部分设置了一个环境变量GET_HOSTS_FROM=env，意思是从环境变量中获取redis-master和redis-slave服务的IP地址信息。

容器镜像名为kubeguide/guestbook-php-frontend，该镜像中所包含的PHP的留言板源码（guestbook.php）如下：

```
< ?
set_include_path('.:usr/local/lib/php');
error_reporting(E_ALL);
ini_set('display_errors', 1);
require 'Predis/Autoloader.php';
Predis\Autoloader::register();

if (isset($_GET['cmd']) === true) {
    $host = 'redis-master';
    if (getenv('GET_HOSTS_FROM') == 'env') {
        $host = getenv('REDIS_MASTER_SERVICE_HOST');
    }
    header('Content-Type: application/json');
    if ($_GET['cmd'] == 'set') {
        $client = new Predis\Client([
```

```

        'scheme' => 'tcp',
        'host'    => $host,
        'port'    => 6379,
    ]);

    $client->set($_GET['key'], $_GET['value']);
    print('{"message": "Updated"}');
} else {
    $host = 'redis-slave';
    if (getenv('GET_HOSTS_FROM') == 'env') {
        $host = getenv('REDIS_SLAVE_SERVICE_HOST');
    }
    $client = new Predis\Client([
        'scheme' => 'tcp',
        'host'    => $host,
        'port'    => 6379,
    ]);

    $value = $client->get($_GET['key']);
    print('{"data": "' . $value . '"}');
}
} else {
    phpinfo();
} ?>

```

这段PHP代码的意思是，如果是一个set请求（提交留言），则会连接到redis-master服务进行写数据操作，其中redis-master服务的虚拟

IP地址是用之前提过的从环境变量中获取的方式得到的，端口使用默认的 6379 端口号（当然，也可以使用环境变量‘REDIS_MASTER_SERVICE_PORT’的值）；如果是get请求，则会连接到redis-slave服务进行读数据操作。

可以看到，如果在容器配置部分不设置env“GET_HOSTS_FROM”，则将使用redis-master或redis-slave服务名来访问这两个服务，这将使用DNS方式的服务发现，需要预先启动Kubernetes集群的skydns服务，详见2.5.4节的说明。

运行kubectl create命令创建RC:

```
$ kubectl create -f frontend-controller.yaml
replicationcontrollers "frontend" created
```

查看已创建的RC:

```
$ kubectl get rc
```

NAME	DESIRED	CURRENT	AGE
frontend	3	3	1h
redis-master	1	1	1h
redis-slave	2	2	1h

再查看生成的Pod:

```
$ kubectl get pods
```

NAME	READY	STATUS
RESTARTS	AGE	

	redis-master-b03io	1/1	Running	
0	1h			
	redis-slave-10ahl	1/1	Running	
0	1h			
	redis-slave-c5y10	1/1	Running	
0	1h			
	frontend-4o11g	1/1	Running	0
1h				
	frontend-u9aq6	1/1	Running	
0	1h			
	frontend-yga1l	1/1	Running	0
1h				

最后创建frontend Service，主要目的是使用Service的NodePort给Kubernetes集群中的Service映射一个外网可以访问的端口，这样一来，外部网络就可以通过NodeIP+NodePort的方式访问集群中的服务了。

服务定义文件frontend-service.yaml的内容如下：

```

apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    name: frontend
spec:
```



```
type: NodePort
ports:
- port: 80
  nodePort: 30001
selector:
  name: frontend
```

这里的关键点是设置`type=NodePort`并指定一个`NodePort`的值，表示使用`Node`上的物理机端口提供对外访问的能力。需要注意的是，`spec.ports.NodePort`的端口号范围可以进行限制（通过`kube-apiserver`的启动参数`--service-node-port-range`指定），默认为30000~32767，如果指定为可用IP范围之外的其他端口号，则`Service`的创建将会失败。

运行`kubectl create`命令创建`Service`:

```
$ kubectl create -f frontend-service.yaml
Services "frontend" created
```

通过`kubectl`查看创建的`Service`:

```
$ kubectl get services
```

	NAME	CLUSTER-IP	EXTERNAL-IP
PORT(S)	AGE		
	frontend	169.169.167.153	<nodes>
80/TCP	25m		
	redis-master	169.169.208.57	<none>
6379/TCP	25m		

redis-slave

169.169.78.102<none>

6379/TCP 25m

2.3.4 通过浏览器访问frontend页面

经过上面的三个步骤就搭建好了Guestbook留言板系统，总共包括3个应用的6个实例，都运行在Kubernetes集群中。打开浏览器，在地址栏输入http://虚拟机IP: 30001/，将看到如图2.6所示的网页，并且看到网页上有一条留言——“Hello World！”。



图2.6 通过浏览器访问留言板网页

尝试输入一条新的留言“Hi Kubernetes！”，单击Submit按钮，网页将会在原留言的下方显示新的留言，说明这条留言已经被成功加入Redis数据库中了，如图2.7所示。

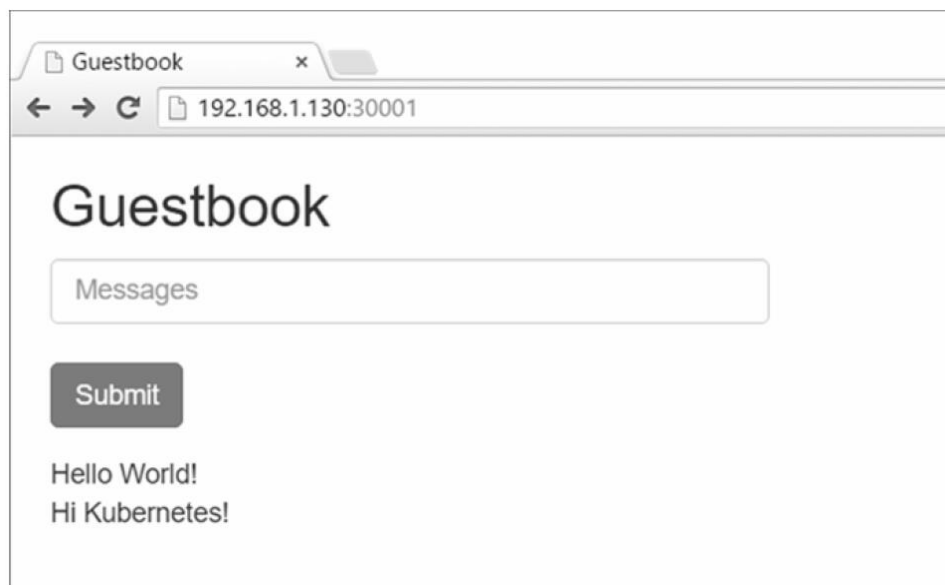


图2.7 在留言板网页添加新的留言

通过Guestbook示例，可以看到Kubernetes强大的应用管理功能，用户仅需通过几个简单的YAML配置就能完成复杂系统的搭建，并能够通过Kubernetes自动实现服务发现和负载均衡。接下来，让我们深入Pod的应用、配置、调度管理及服务的应用，开始Kubernetes应用管理之旅。

2.4 深入掌握Pod

本节将对Kubernetes如何发布和管理应用进行详细说明和示例，主要包括Pod和容器的使用、Pod的控制和调度管理、应用配置管理等内容。

2.4.1 Pod定义详解

yaml格式的Pod定义文件的完整内容如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: string
  namespace: string
  labels:
    - name: string
  annotations:
    - name: string
spec:
  containers:
    - name: string
      image: string
      imagePullPolicy: [Always | Never | IfNotPresent]
      command: [string]
  args: [string]
  workingDir: string
  volumeMounts:
    - name: string
      mountPath: string
```

```
    readOnly: boolean
  ports:
    - name: string
      containerPort: int
      hostPort: int
      protocol: string
  env:
    - name: string
      value: string
  resources:
    limits:
      cpu: string
      memory: string
  requests:
    cpu: string
    memory: string
  livenessProbe:
    exec:
      command: [string]
    httpGet:
      path: string
      port: number
      host: string
      scheme: string
      httpHeaders:
        - name: string
          value: string
```

```

    tcpSocket:
      port: number
      initialDelaySeconds: 0
      timeoutSeconds: 0
      periodSeconds: 0
      successThreshold: 0
      failureThreshold: 0
    securityContext:
      privileged: false
    restartPolicy: [Always | Never | OnFailure]
    nodeSelector: object
    imagePullSecrets:
-   name: string
    hostNetwork: false
    volumes:
-   name: string
      emptyDir: {}
      hostPath:
        path: string
      secret:
        secretName: string
        items:
-         key: string
          path: string
    configMap:
      name: string
      items:

```


- key: string
path: string

对各属性的详细说明如表2.13所示。

表2.13 对Pod定义文件模板中各属性的详细说明

属 性 名 称	取 值 类 型	是 否 必 选	取 值 说 明
version	String	Required	版本号，例如 v1
kind	String	Required	Pod
metadata	Object	Required	元数据
metadata.name	String	Required	Pod 的名称，命名规范需符合 RFC 1035 规范
metadata.namespace	String	Required	Pod 所属的命名空间，默认为 “default”
metadata.labels[]	List		自定义标签列表
metadata.annotation[]	List		自定义注解列表
Spec	Object	Required	Pod 中容器的详细定义
spec.containers[]	List	Required	Pod 中的容器列表
spec.containers[].name	String	Required	容器的名称，需符合 RFC 1035 规范
spec.containers[].image	String	Required	容器的镜像名称

续表

属性名称	取值类型	是否必选	取值说明
spec.containers[].imagePullPolicy	String		获取镜像的策略，可选值包括：Always、Never、IfNotPresent，默认值为 Always。 Always：表示每次都尝试重新下载镜像。 IfNotPresent：表示如果本地有该镜像，则使用本地的镜像，本地不存在时下载镜像。 Never：表示仅使用本地镜像
spec.containers[].command[]	List		容器的启动命令列表，如果不指定，则使用镜像打包时使用的启动命令
spec.containers[].args[]	List		容器的启动命令参数列表
spec.containers[].workingDir	String		容器的工作目录
spec.containers[].volumeMounts[]	List		挂载到容器内部的存储卷配置
spec.containers[].volumeMounts[].name	String		引用 Pod 定义的共享存储卷的名称，需使用 volumes[] 部分定义的共享存储卷名称
spec.containers[].volumeMounts[].mountPath	String		存储卷在容器内 Mount 的绝对路径，应少于 512 个字符
spec.containers[].volumeMounts[].readOnly	Boolean		是否为只读模式，默认为读写模式
spec.containers[].ports[]	List		容器需要暴露的端口号列表
spec.containers[].ports[].name	String		端口的名称
spec.containers[].ports[].containerPort	Int		容器需要监听的端口号
spec.containers[].ports[].hostPort	Int		容器所在主机需要监听的端口号，默认与 containerPort 相同。设置 hostPort 时，同一台宿主主机将无法启动该容器的第 2 份副本
spec.containers[].ports[].protocol	String		端口协议，支持 TCP 和 UDP，默认为 TCP
spec.containers[].env[]	List		容器运行前需设置的环境变量列表
spec.containers[].env[].name	String		环境变量的名称
spec.containers[].env[].value	String		环境变量的值
spec.containers[].resources	Object		资源限制和资源请求的设置，详见第 5 章的说明
spec.containers[].resources.limits	Object		资源限制的设置
spec.containers[].resources.limits.cpu	String		CPU 限制，单位为 core 数，将用于 docker run --cpu-shares 参数
spec.containers[].resources.limits.memory	String		内存限制，单位可以为 MiB/GiB 等，将用于 docker run --memory 参数
spec.containers[].resources.requests	Object		资源限制的设置

续表

属性名称	取值类型	是否必选	取值说明
<code>spec.containers[].resources.requests.cpu</code>	String		CPU 请求，单位为 core 数，容器启动的初始可用数量
<code>spec.containers[].resources.requests.memory</code>	String		内存请求，单位可以为 MiB、GiB 等，容器启动的初始可用数量
<code>spec.volumes[]</code>	List		在该 Pod 上定义的共享存储卷列表
<code>spec.volumes[].name</code>	String		共享存储卷的名称，在一个 Pod 中每个存储卷定义一个名称，应符合 RFC 1035 规范。容器定义部分的 <code>containers[].volumeMounts[].name</code> 将引用该共享存储卷的名称。 volume 的类型包括： <code>emptyDir</code> 、 <code>hostPath</code> 、 <code>gcePersistentDisk</code> 、 <code>awsElasticBlockStore</code> 、 <code>gitRepo</code> 、 <code>secret</code> 、 <code>nfs</code> 、 <code>iscsi</code> 、 <code>glusterfs</code> 、 <code>persistentVolumeClaim</code> 、 <code>rbd</code> 、 <code>flexVolume</code> 、 <code>cinder</code> 、 <code>cephfs</code> 、 <code>flocker</code> 、 <code>downwardAPI</code> 、 <code>fc</code> 、 <code>azureFile</code> 、 <code>configMap</code> 、 <code>vsphereVolume</code> ，可以定义多个 volume，每个 volume 的 name 保持唯一。本节讲解 <code>emptyDir</code> 、 <code>hostPath</code> 、 <code>secret</code> 、 <code>configMap</code> 这 4 种 volume，其他类型 volume 的设置方式详见第 1 章的说明
<code>spec.volumes[].emptyDir</code>	Object		类型为 <code>emptyDir</code> 的存储卷，表示与 Pod 同生命周期的一个临时目录，其值为一个空对象： <code>emptyDir: {}</code>
<code>spec.volumes[].hostPath</code>	Object		类型为 <code>hostPath</code> 的存储卷，表示挂载 Pod 所在宿主机的目录，通过 <code>volumes[].hostPath.path</code> 指定
<code>spec.volumes[].hostPath.path</code>	String		Pod 所在主机的目录，将被用于容器中 <code>mount</code> 的目录
<code>spec.volumes[].secret</code>	Object		类型为 <code>secret</code> 的存储卷，表示挂载集群预定义的 <code>secret</code> 对象到容器内部
<code>spec.volumes[].configMap</code>	Object		类型为 <code>configMap</code> 的存储卷，表示挂载集群预定义的 <code>configMap</code> 对象到容器内部
<code>spec.volumes[].livenessProbe</code>	Object		对 Pod 内各容器健康检查的设置，当探测无响应几次之后，系统将自动重启该容器。可以设置的方法包括： <code>exec</code> 、 <code>httpGet</code> 和 <code>tcpSocket</code> 。对一个容器仅需设置一种健康检查方法
<code>spec.volumes[].livenessProbe.exec</code>	Object		对 Pod 内各容器健康检查的设置， <code>exec</code> 方式
<code>spec.volumes[].livenessProbe.exec.command[]</code>	String		<code>exec</code> 方式需要指定的命令或者脚本
<code>spec.volumes[].livenessProbe.httpGet</code>	Object		对 Pod 内各容器健康检查的设置， <code>HTTPGet</code> 方式。需指定 <code>path</code> 、 <code>port</code>

续表

属 性 名 称	取 值 类 型	是 否 必 选	取 值 说 明
spec.volumes[].livenessProbe.tcpSocket	Object		对 Pod 内各容器健康检查的设置，tcpSocket 方式
spec.volumes[].livenessProbe.initialDelaySeconds	Number		容器启动完成后进行首次探测的时间，单位为秒
spec.volumes[].livenessProbe.timeoutSeconds	Number		对容器健康检查的探测等待响应的超时时间设置，单位为秒，默认为 1 秒。超过该超时时间设置，将认为该容器不健康，将重启该容器
spec.volumes[].livenessProbe.periodSeconds	Number		对容器健康检查的定期探测时间设置，单位为秒，默认为 10 秒探测一次
spec.restartPolicy	String		Pod 的重启策略，可选值为 Always、OnFailure，默认值为 Always。 Always: Pod 一旦终止运行，则无论容器是如何终止的，kubelet 都将重启它。 OnFailure: 只有 Pod 以非零退出码终止时，kubelet 才会重启该容器。如果容器正常结束（退出码为 0），则 kubelet 将不会重启它。 Never: Pod 终止后，kubelet 将退出码报告给 Master，不会再重启该 Pod
spec.nodeSelector	Object		设置 NodeSelector 表示将该 Pod 调度到包含这些 label 的 Node 上，以 key:value 格式指定
spec.imagePullSecrets	Object		Pull 镜像时使用的 secret 名称，以 name:secretkey 格式指定
spec.hostNetwork	Boolean		是否使用主机网络模式，默认为 false。如果设置为 true，则表示容器使用宿主机网络，不再使用 Docker 网桥，该 Pod 将无法在同一台宿主机上启动第 2 个副本

2.4.2 Pod的基本用法

在对Pod的用法进行说明之前，有必要先对Docker容器中应用的运行要求进行说明。

在使用Docker时，可以使用`docker run`命令创建并启动一个容器。而在Kubernetes系统对长时间运行容器的要求是：其主程序需要一直在前台执行。如果我们创建的Docker镜像的启动命令是后台执行程序，例如Linux脚本：

```
nohup ./start.sh &
```

则在kubelet创建包含这个容器的Pod之后运行完该命令，即认为Pod执行结束，将立刻销毁该Pod。如果为该Pod定义了ReplicationController，则系统将会监控到该Pod已经终止，之后根据RC定义中Pod的replicas副本数量生成一个新的Pod。而一旦创建出新的Pod，就将在执行完启动命令后，陷入无限循环的过程中。这就是Kubernetes需要我们自己创建的Docker镜像以一个前台命令作为启动命令的原因。

对于无法改造为前台执行的应用，也可以使用开源工具Supervisor辅助进行前台运行的功能。Supervisor提供了一种可以同时启动多个后台应用，并保持Supervisor自身在前台执行的机制，可以满足Kubernetes对容器的启动要求。关于Supervisor的安装和使用，请参考官网<http://supervisord.org>的文档说明。

接下来对Pod对容器的封装和应用进行说明，Pod的基本用法为：Pod可以由1个或多个容器组合而成。

在上一节Guestbook的例子中，名为frontend的Pod只由一个容器组成：

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  containers:
    - name: frontend
      image: kubeguide/guestbook-php-frontend
      env:
        - name: GET_HOSTS_FROM
          value: env
      ports:
        - containerPort: 80
```

这个frontendPod在成功启动之后，将启动1个Docker容器。

另一种场景是，当frontend和redis两个容器应用为紧耦合的关系，应该组合成一个整体对外提供服务时，则应将这两个容器打包为一个Pod，如图2.8所示。

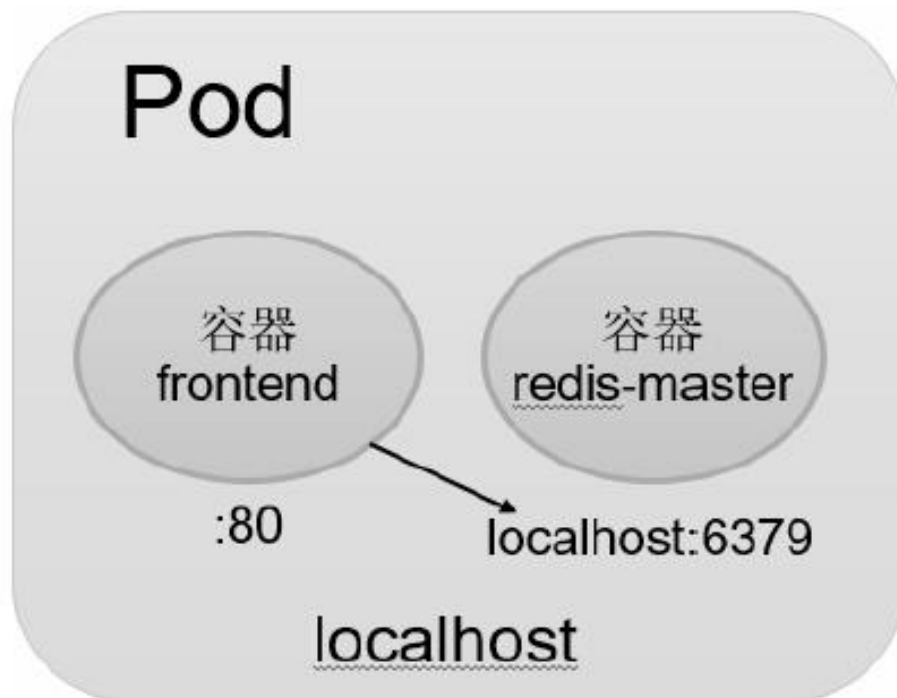


图2.8 包含两个容器的Pod

配置文件frontend-localredis-pod.yaml如下:

```
apiVersion: v1
kind: Pod
metadata:
  name: redis-php
  labels:
    name: redis-php
spec:
  containers:
    - name: frontend
      image: kubeguide/guestbook-php-frontend:localredis
```

```
ports:
  - containerPort: 80
- name: redis
  image: kubeguide/redis-master
  ports:
    - containerPort: 6379
```

属于一个Pod的多个容器应用之间相互访问时仅需要通过localhost就可以通信，使得这一组容器被“绑定”在了一个环境中。

在Docker容器kubeguide/guestbook-php-frontend: localredis的PHP网页中，直接通过URL地址“localhost: 6379”对同属于一个Pod内的redis-master进行访问。guestbook.php的内容如下：

```
< ?
set_include_path('.:usr/local/lib/php');
error_reporting(E_ALL);
ini_set('display_errors', 1);
require 'Predis/Autoloader.php';
Predis\Autoloader::register();

if (isset($_GET['cmd']) === true) {
    $host = 'localhost';
    if (getenv('REDIS_HOST') &&
strlen(getenv('REDIS_HOST')) > 0 ) {
        $host = getenv('REDIS_HOST');
    }
}
```



```

header('Content-Type: application/json');
if ($_GET['cmd'] == 'set') {
    $client = new Predis\Client([
        'scheme' => 'tcp',
        'host'    => $host,
        'port'    => 6379,
    ]);

    $client->set($_GET['key'], $_GET['value']);
    print('{"message": "Updated"}');
} else {
    $host = 'localhost';

    if (getenv('REDIS_HOST') &&
strlen(getenv('REDIS_HOST')) > 0 ) {
        $host = getenv('REDIS_HOST');
    }

    $client = new Predis\Client([
        'scheme' => 'tcp',
        'host'    => $host,
        'port'    => 6379,
    ]);

    $value = $client->get($_GET['key']);
    print('{"data": "' . $value . '"}');
}
} else {

```

```
    phpinfo();  
} ? >
```

运行kubect1 create命令创建该Pod:

```
$ kubect1 create -f frontend-localredis-pod.yaml  
pod "redis-php" created
```

查看已创建的Pod:

```
# kubect1 get pods  
  
NAME           READY    STATUS    RESTARTS   AGE  
redis-php      2/2      Running   0           10m
```

可以看到READY信息为2/2，表示Pod中的两个容器都成功运行了。

查看这个Pod的详细信息，可以看到两个容器的定义及创建的过程（Event事件信息）：

```
# kubect1 describe pod redis-php  
  
Name:          redis-php  
Namespace:     default  
Node:          k8s/192.168.18.3  
Start Time:    Thu, 28 Jul 2016 12:28:21 +0800  
Labels:        name=redis-php  
Status:        Running  
IP:            172.17.1.4
```

Controllers: <none>

Containers:

frontend:

Container ID:

docker://ccc8616f8df1fb19abbd0ab189a36e6f6628b78ba7b97b1077
d86e7fc224ee08

Image: kubeguide/guestbook-
php-frontend:localredis

Image ID:

docker://sha256:d014f67384a11186e135b95a7ed0d794674f7ce258f
0dce47267c3052a0d0fa9

Port: 80/TCP

State: Running

Started: Thu, 28 Jul 2016
12:28:22 +0800

Ready: True

Restart Count: 0

Environment Variables: <none>

redis:

Container ID:

docker://c0b19362097cda6dd5b8ed7d8eaaaf43aeeb969ee023ef2556
04bde089808075

Image: kubeguide/redis-master

Image ID:

docker://sha256:405a0b586f7ebef545ec65be0e914311159d1baedcc
d3a93e9d3e3b249ec5cbd

Port: 6379/TCP

```

State: Running
Started: Thu, 28 Jul 2016
12:28:23 +0800
Ready: True
Restart Count: 0
Environment Variables: <none>

```

Conditions:

Type	Status
Initialized	True
Ready	True
PodScheduled	True

Volumes:

default-token-97j21:

Type: Secret (a volume populated by a Secret)

SecretName: default-token-97j21

QoS Tier: BestEffort

Events:

	FirstSeen	LastSeen	Count	From	SubobjectPath
Type	Reason	Message			
	-----	-----	-----	----	-----
	-----	-----			
	18m18m		1	{default-scheduler }	
Normal		Scheduled		Successfully assigned redis-	
		php to k8s-node-1			
	18m18m		1	{kubelet k8s-node-1}	
		spec.containers{frontend}		Normal	Pulled

```
Container          image          "kubeguide/guestbook-php-frontend:localredis" already present on machine
18m18m            1            {kubelet k8s-node-1}
spec.containers{frontend}      Normal      Created
Created container with docker id ccc8616f8df1
18m18m            1            {kubelet k8s-node-1}
spec.containers{frontend}      Normal      Started
Started container with docker id ccc8616f8df1
18m18m            1            {kubelet k8s-node-1}
spec.containers{redis}         Normal      Pulled
Container image "kubeguide/redis-master" already present on machine
18m18m            1            {kubelet k8s-node-1}
spec.containers{redis}         Normal      Created
Created container with docker id c0b19362097c
18m18m            1            {kubelet k8s-node-1}
spec.containers{redis}         Normal      Started
Started container with docker id c0b19362097c
```

2.4.3 静态Pod

静态Pod是由kubelet进行管理的仅存在于特定Node上的Pod。它们不能通过API Server进行管理，无法与ReplicationController、Deployment或者DaemonSet进行关联，并且kubelet也无法对它们进行健康检查。静态Pod总是由kubelet进行创建，并且总是在kubelet所在的Node上运行。

创建静态Pod有两种方式：配置文件或者HTTP方式。

1) 配置文件方式

首先，需要设置kubelet的启动参数“--config”，指定kubelet需要监控的配置文件所在的目录，kubelet会定期扫描该目录，并根据该目录中的.yaml或.json文件进行创建操作。

假设配置目录为 /etc/kubelet.d/，配置启动参数：--config=/etc/kubelet.d/，然后重启kubelet服务。

在目录/etc/kubelet.d中放入static-web.yaml文件，内容如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: static-web
  labels:
```

```

        name: static-web
spec:
  containers:
  - name: static-web
    image: nginx
    ports:
    - name: web
      containerPort: 80

```

等待一会儿，查看本机中已经启动的容器：

```

# docker ps
CONTAINER ID   IMAGE    COMMAND                  CREATED        STATUS
PORTS          NAMES
2292ea231ab1   nginx    "nginx -g 'daemon off'" 1 minute ago   1m
k8s_static-web.68ee0075_static-web-k8s-
node-1_default_78c7efddeb191c949cbb7aa22a927c8_401b96d0

```

可以看到一个Nginx容器已经被kubelet成功创建了出来。

到Master节点查看Pod列表，可以看到这个static pod：

```

# kubectl get pods
NAME                                READY   STATUS
RESTARTS   AGE
static-web-node1    1/1     Running
5m

```

由于静态Pod无法通过API Server直接管理，所以在Master节点尝试删除这个Pod，将会使其变成Pending状态，且不会被删除。

```
# kubectl delete pod static-web-node1
pod "static-web-node1" deleted

# kubectl get pods
```

	NAME	READY	STATUS	RESTARTS
AGE				
	static-web-node1	0/1	Pending	0
1s				

删除该Pod的操作只能是到其所在Node上，将其定义文件static-web.yaml从/etc/kubelet.d目录下删除。

```
# rm /etc/kubelet.d/static-web.yaml
# docker ps
// 无容器正在运行。
```

2.4.4 Pod容器共享Volume

在同一个Pod中的多个容器能够共享Pod级别的存储卷Volume。Volume可以定义为各种类型，多个容器各自进行挂载操作，将一个Volume挂载为容器内部需要的目录，如图2.9所示。

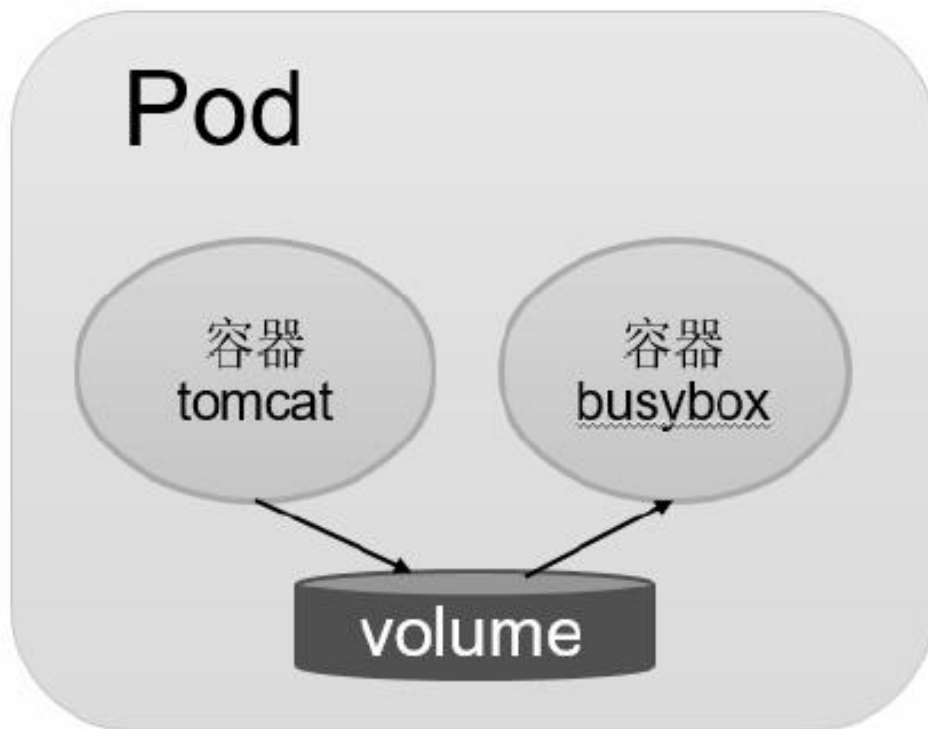


图2.9 Pod中多个容器共享volume

在下面的例子中，Pod内包含两个容器：tomcat和busybox，在Pod级别设置Volume“app-logs”，用于tomcat向其中写日志文件，busybox读日志文件。

配置文件pod-volume-applogs.yaml的内容如下:

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-pod
spec:
  containers:
    - name: tomcat
      image: tomcat
      ports:
        - containerPort: 8080
      volumeMounts:
        - name: app-logs
          mountPath: /usr/local/tomcat/logs
    - name: busybox
      image: busybox
      command: ["sh", "-c", "tail -f
/logs/catalina*.log"]
      volumeMounts:
        - name: app-logs
          mountPath: /logs
  volumes:
    - name: app-logs
      emptyDir: {}
```

这里设置的Volume名为app-logs，类型为emptyDir（也可以设置为其他类型，详见第1章对Volume概念的说明），挂载到tomcat容器内的/usr/local/tomcat/logs目录，同时挂载到logreader容器内的/logs目录。tomcat容器在启动后会向/usr/local/tomcat/logs目录中写文件，logreader容器就可以读取其中的文件了。

logreader容器的启动命令为tail-f/logs/catalina*.log，我们可以通过kubectl logs命令查看logreader容器的输出内容：

```
# kubectl logs volume-pod -c busybox

.....

29-Jul-2016 12:55:59.626 INFO [localhost-startStop-1]
org.apache.catalina.startup.HostConfig.deployDirectory
Deploying          web          application          directory
/usr/local/tomcat/webapps/manager

29-Jul-2016 12:55:59.722 INFO [localhost-startStop-1]
org.apache.catalina.startup.HostConfig.deployDirectory
Deployment      of      web      application      directory
/usr/local/tomcat/webapps/manager has finished in 96 ms

29-Jul-2016 12:55:59.740 INFO [main]
org.apache.coyote.AbstractProtocol.start          Starting
ProtocolHandler ["http-apr-8080"]

29-Jul-2016 12:55:59.794 INFO [main]
org.apache.coyote.AbstractProtocol.start          Starting
ProtocolHandler ["ajp-apr-8009"]

29-Jul-2016 12:56:00.604 INFO [main]
```

```
org.apache.catalina.startup.Catalina.start Server startup
in 4052 ms
```

这个文件即为tomcat生成的日志文件/usr/local/tomcat/logs/catalina.<date>.log的内容。登录tomcat容器进行查看:

```
# kubectl exec -ti volume-pod -c tomcat -- ls
/usr/local/tomcat/logs
                                catalina.2016-07-29.log
localhost_access_log.2016-07-29.txt
                                host-manager.2016-07-29.log  manager.2016-07-29.log

# kubectl exec -ti volume-pod -c tomcat -- tail
/usr/local/tomcat/logs/catalina.2016-07-29.log
.....
29-Jul-2016 12:55:59.722 INFO [localhost-startStop-1]
org.apache.catalina.startup.HostConfig.deployDirectory
Deployment of web application directory
/usr/local/tomcat/webapps/manager has finished in 96 ms
                29-Jul-2016 12:55:59.740 INFO [main]
org.apache.coyote.AbstractProtocol.start Starting
ProtocolHandler ["http-apr-8080"]
                29-Jul-2016 12:55:59.794 INFO [main]
org.apache.coyote.AbstractProtocol.start Starting
ProtocolHandler ["ajp-apr-8009"]
                29-Jul-2016 12:56:00.604 INFO [main]
```

org.apache.catalina.startup.Catalina.start Server startup
in 4052 ms

2.4.5 Pod的配置管理

应用部署的一个最佳实践是将应用所需的配置信息与程序进行分离，这样可以使得应用程序被更好地复用，通过不同的配置也能实现更灵活的功能。将应用打包为容器镜像后，可以通过环境变量或者外挂文件的方式在创建容器时进行配置注入，但在大规模容器集群的环境中，对多个容器进行不同的配置将变得非常复杂。Kubernetesv1.2版本提供了一种统一的集群配置管理方案——ConfigMap。本节对ConfigMap的概念和用法进行详细描述。

1.ConfigMap: 容器应用的配置管理

ConfigMap供容器使用的典型用法如下。

- (1) 生成为容器内的环境变量。
- (2) 设置容器启动命令的启动参数（需设置为环境变量）。
- (3) 以Volume的形式挂载为容器内部的文件或目录。

ConfigMap以一个或多个key: value的形式保存在Kubernetes系统中供应用使用，既可以用于表示一个变量的值（例如apploglevel=info），也可以用于表示一个完整配置文件的内容（例如server.xml=<? xml...>...）

可以通过yaml配置文件或者直接使用kubect create configmap命令行的方式来创建ConfigMap。

2.ConfigMap的创建: yaml文件方式

下面的例子cm-appvars.yaml描述了将几个应用所需的变量定义为ConfigMap的用法:

```
cm-appvars.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cm-appvars
data:
  apploglevel: info
  appdatadir: /var/data
```

执行kubect create命令创建该ConfigMap:

```
$kubectl create -f cm-appvars.yaml
configmap "cm-appvars" created
```

查看创建好的ConfigMap:

```
# kubectl get configmap
```

NAME	DATA	AGE
cm-appvars	2	3s

```
# kubectl describe configmap cm-appvars
```

```
Name:          cm-appvars
```

```
Namespace:     default
```

```
Labels:        <none>
```

```
Annotations:    <none>
```

```
Data
```

```
====
```

```
appdatadir:    9 bytes
```

```
apploglevel:   4 bytes
```

```
# kubectl get configmap cm-appvars -o yaml
```

```
apiVersion: v1
```

```
data:
```

```
  appdatadir: /var/data
```

```
  apploglevel: info
```

```
kind: ConfigMap
```

```
metadata:
```

```
  creationTimestamp: 2016-07-28T19:57:16Z
```

```
  name: cm-appvars
```

```
  namespace: default
```

```
  resourceVersion: "78709"
```

```
  selfLink: /api/v1/namespaces/default/configmaps/cm-
```

```
appvars
```

```
  uid: 7bb2e9c0-54fd-11e6-9dcd-000c29dc2102
```

下面的例子 `cm-appconfigfiles.yaml` 描述了将两个配置文件 `server.xml` 和 `logging.properties` 定义为 `ConfigMap` 的用法，设置 `key` 为配置文件的别名，`value` 则是配置文件的全部文本内容：

```
cm-appconfigfiles.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cm-appconfigfiles
data:
  key-serverxml: |
    < xml version='1.0' encoding='utf-8' >
    <Server port="8005" shutdown="SHUTDOWN">
                                     <Listener
className="org.apache.catalina.startup.VersionLoggerListene
r" />
                                     <Listener
className="org.apache.catalina.core.AprLifecycleListener"
SSLEngine="on" />
    <Listener className=
"org.apache.catalina.core.JreMemoryLeakPreventionListener"
/>
    <Listener className=
"org.apache.catalina.mbeans.GlobalResourcesLifecycleListene
r" />
    <Listener className=
"org.apache.catalina.core.ThreadLocalLeakPreventionListener
```

```

" />
    <GlobalNamingResources>
        <Resource name="UserDatabase" auth="Container"

type="org.apache.catalina.UserDatabase"
                description="User database that can
be updated and saved"

factory="org.apache.catalina.users.MemoryUserDatabaseFactor
y"
                pathname="conf/tomcat-users.xml" />
    </GlobalNamingResources>

    <Service name="Catalina">
        <Connector port="8080" protocol="HTTP/1.1"
                connectionTimeout="20000"
                redirectPort="8443" />
        <Connector port="8009" protocol="AJP/1.3"
redirectPort="8443" />
        <Engine name="Catalina" defaultHost="localhost">
                                <Realm
className="org.apache.catalina.realm.LockOutRealm">
                                <Realm
className="org.apache.catalina.realm.UserDatabaseRealm"
                                resourceName="UserDatabase"/>
        </Realm>
        <Host name="localhost" appBase="webapps"

```

```

        unpackWARs="true" autoDeploy="true">
            <Valve
className="org.apache.catalina.valves.AccessLogValve"
directory="logs"
                prefix="localhost_access_log"
suffix=".txt"
                pattern="%h %l %u %t &quot;%r&quot;
%s %b" />

```

```

    </Host>

```

```

</Engine>

```

```

</Service>

```

```

</Server>

```

```

    key-loggingproperties: "handlers
                        =1catalina.org.apache.juli.FileHandler,
2localhost.org.apache.juli.FileHandler,
                        3manager.org.apache.juli.FileHandler, 4host-
manager.org.apache.juli.FileHandler,
                        java.util.logging.ConsoleHandler\r\n\r\n.handlers=
1catalina.org.apache.juli.FileHandler,

```

```

java.util.logging.ConsoleHandler\r\n\r\n1catalina.org.apach
e.juli.FileHandler.level
                        =
FINE\r\n1catalina.org.apache.juli.FileHandler.directory    =
${catalina.base}/logs\r\n1catalina.org.apache.juli.FileHand
ler.prefix

```

```

=
catalina.\r\n\r\n2localhost.org.apache.juli.FileHandler.level
=
FINE\r\n2localhost.org.apache.juli.FileHandler.directory
=
${catalina.base}/logs\r\n2localhost.org.apache.juli.FileHandler.prefix
=
localhost.\r\n\r\n3manager.org.apache.juli.FileHandler.level
=
FINE\r\n3manager.org.apache.juli.FileHandler.directory
=
${catalina.base}/logs\r\n3manager.org.apache.juli.FileHandler.prefix
=
manager.\r\n\r\n4host-
manager.org.apache.juli.FileHandler.level = FINE\r\n4host-
manager.org.apache.juli.FileHandler.directory
= ${catalina.base}/logs\r\n4host-
manager.org.apache.juli.FileHandler.prefix =
host-
manager.\r\n\r\njava.util.logging.ConsoleHandler.level =
FINE\r\njava.util.logging.ConsoleHandler.formatter
=
java.util.logging.SimpleFormatter\r\n\r\n\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].level
= INFO\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].handlers
=

```

```

2localhost.org.apache.juli.FileHandler\r\n\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
[/manager].level
    = INFO\r\norg.apache.catalina.core.ContainerBase.
[Catalina].[localhost].[/manager].handlers
=
3manager.org.apache.juli.FileHandler\r\n\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/host-
manager].level
    = INFO\r\norg.apache.catalina.core.ContainerBase.
[Catalina].[localhost].[/host-manager].handlers
=      4host-
manager.org.apache.juli.FileHandler\r\n\r\n"

```

执行kubect create命令创建该ConfigMap:

```

$kubectl create -f cm-appconfigfiles.yaml
configmap "cm-appconfigfiles" created

```

查看创建好的ConfigMap:

```

# kubectl get configmap cm-appconfigfiles
NAME          DATA      AGE
cm-appconfigfiles  2          14s

# kubectl describe configmap cm-appconfigfiles
Name:          cm-appconfigfiles

```

Namespace: default
Labels: <none>
Annotations: <none>

Data

====

key-loggingproperties: 1809 bytes
key-serverxml: 1686 bytes

查看已创建的ConfigMap的详细内容，可以看到两个配置文件的全文：

```
# kubectl get configmap cm-appconfigfiles -o yaml
apiVersion: v1
data:
    key-loggingproperties: "handlers =
1catalina.org.apache.juli.FileHandler,
2localhost.org.apache.juli.FileHandler,
3manager.org.apache.juli.FileHandler, 4host-
manager.org.apache.juli.FileHandler,
    java.util.logging.ConsoleHandler\r\n\r\n.handlers
= 1catalina.org.apache.juli.FileHandler,
    java.util.logging.ConsoleHandler\r\n\r\n1catalina.org.apach
e.juli.FileHandler.level
=
FINE\r\n1catalina.org.apache.juli.FileHandler.directory =
```

```

${catalina.base}/logs\r\n1catalina.org.apache.juli.FileHand
ler.prefix
=
catalina.\r\n\r\n2localhost.org.apache.juli.FileHandler.lev
el
=
FINE\r\n2localhost.org.apache.juli.FileHandler.directory
=
${catalina.base}/logs\r\n2localhost.org.apache.juli.FileHan
dler.prefix
=
localhost.\r\n\r\n3manager.org.apache.juli.FileHandler.leve
l
=
FINE\r\n3manager.org.apache.juli.FileHandler.directory
=
${catalina.base}/logs\r\n3manager.org.apache.juli.FileHandl
er.prefix
=
manager.\r\n\r\n4host-
manager.org.apache.juli.FileHandler.level = FINE\r\n4host-
manager.org.apache.juli.FileHandler.directory
=
${catalina.base}/logs\r\n4host-
manager.org.apache.juli.FileHandler.prefix =
host-
manager.\r\n\r\njava.util.logging.ConsoleHandler.level
=
FINE\r\njava.util.logging.ConsoleHandler.formatter
=
java.util.logging.SimpleFormatter\r\n\r\n\r\norg.apache.cat
alina.core.ContainerBase.[Catalina].[localhost].level
= INFO\r\norg.apache.catalina.core.ContainerBase.

```

```

[Catalina].[localhost].handlers
=
2localhost.org.apache.juli.FileHandler\r\n\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
[/manager].level
= INFO\r\norg.apache.catalina.core.ContainerBase.
[Catalina].[localhost].[/manager].handlers
=
3manager.org.apache.juli.FileHandler\r\n\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/host-
manager].level
= INFO\r\norg.apache.catalina.core.ContainerBase.
[Catalina].[localhost].[/host-manager].handlers
= 4host-
manager.org.apache.juli.FileHandler\r\n\r\n"
key-serverxml: |
< xml version='1.0' encoding='utf-8' >
<Server port="8005" shutdown="SHUTDOWN">
<Listener
className="org.apache.catalina.startup.VersionLoggerListene
r" />
<Listener
className="org.apache.catalina.core.AprLifecycleListener"
SSLEngine="on" />
<Listener
className="org.apache.catalina.core.JreMemoryLeakPrevention
Listener" />

```



```
<Listener
className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />
```

```
<Listener
className="org.apache.catalina.core.ThreadLocalLeakPreventionListener" />
```

```
<GlobalNamingResources>
```

```
<Resource name="UserDatabase" auth="Container"
```

```
type="org.apache.catalina.UserDatabase"
```

```
description="User database that can
be updated and saved"
```

```
factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
```

```
pathname="conf/tomcat-users.xml" />
```

```
</GlobalNamingResources>
```

```
<Service name="Catalina">
```

```
<Connector port="8080" protocol="HTTP/1.1"
```

```
connectionTimeout="20000"
```

```
redirectPort="8443" />
```

```
<Connector port="8009" protocol="AJP/1.3"
redirectPort="8443" />
```

```
<Engine name="Catalina" defaultHost="localhost">
```

```
<Realm
```

```
className="org.apache.catalina.realm.LockOutRealm">
```

```

<Realm
className="org.apache.catalina.realm.UserDatabaseRealm"
resourceName="UserDatabase"/>
</Realm>
<Host name="localhost" appBase="webapps"
unpackWARs="true" autoDeploy="true">
<Valve
className="org.apache.catalina.valves.AccessLogValve"
directory="logs"
prefix="localhost_access_log"
suffix=".txt"
pattern="%h %l %u %t &quot;%r&quot;
%s %b" />

```

```

</Host>
</Engine>
</Service>
</Server>
kind: ConfigMap
metadata:
  creationTimestamp: 2016-07-29T00:52:18Z
  name: cm-appconfigfiles
  namespace: default
  resourceVersion: "85054"
  selfLink: /api/v1/namespaces/default/configmaps/cm-
appconfigfiles
  uid: b30d5019-5526-11e6-9dcd-000c29dc2102

```

3.ConfigMap的创建: kubectl命令行方式

不使用yaml文件，直接通过kubectl create configmap也可以创建ConfigMap，可以使用参数--from-file或--from-literal指定内容，并且可以在一行命令中指定多个参数。

(1) 通过--from-file参数从文件中进行创建，可以指定key的名称，也可以在一个命令行中创建包含多个key的ConfigMap，语法为：

```
# kubectl create configmap NAME --from-file=[key=]source --from-file=[key=]source
```

(2) 通过--from-file参数从目录中进行创建，该目录下的每个配置文件名都被设置为key，文件的内容被设置为value，语法为：

```
# kubectl create configmap NAME --from-file=config-files-dir
```

(3) --from-literal从文本中进行创建，直接将指定的key#=value#创建为ConfigMap的内容，语法为：

```
# kubectl create configmap NAME --from-literal=key1=value1 --from-literal=key2=value2
```

下面对这几种用法举例说明。

例如，当前目录下含有配置文件`server.xml`，可以创建一个包含该文件内容的`ConfigMap`：

```
# kubectl create configmap cm-server.xml --from-
file=server.xml
configmap "cm-server.xml" created

# kubectl describe configmap cm-server.xml
Name:          cm-server.xml
Namespace:     default
Labels:        <none>
Annotations:   <none>

Data
====
server.xml:    6458 bytes
```

假设 `configfiles` 目录下包含两个配置文件 `server.xml` 和 `logging.properties`，创建一个包含这两个文件内容的`ConfigMap`：

```
# kubectl create configmap cm-appconf --from-
file=configfiles
configmap "cm-appconf" created

# kubectl describe configmap cm-appconf
Name:          cm-appconf
Namespace:     default
```

```
Labels:          <none>
Annotations:     <none>

Data
====
logging.properties:    3354 bytes
server.xml:            6458 bytes
```

使用--from-literal参数进行创建的示例如下:

```
# kubectl create configmap cm-appenv --from-
literal=loglevel=info --from-literal=appdatadir=/var/data
configmap "cm-appenv" created

# kubectl describe configmap cm-appenv
Name:          cm-appenv
Namespace:     default
Labels:        <none>
Annotations:   <none>

Data
====
appdatadir:    9 bytes
loglevel:      4 bytes
```

容器应用对ConfigMap的使用有以下两种方法。

(1) 通过环境变量获取ConfigMap中的内容。

(2) 通过Volume挂载的方式将ConfigMap中的内容挂载为容器内部的文件或目录。

4.ConfigMap的使用：环境变量方式

以cm-appvars.yaml为例：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cm-appvars
data:
  apploglevel: info
  appdatadir: /var/data
```

在Pod“cm-test-pod”的定义中，将ConfigMap“cm-appvars”中的内容以环境变量（APPLOGLEVEL和APPDATADIR）设置为容器内部的环境变量，容器的启动命令将显示这两个环境变量的值（“env|grep APP”）：

```
apiVersion: v1
kind: Pod
metadata:
  name: cm-test-pod
spec:
```

```

containers:
- name: cm-test
  image: busybox
  command: [ "/bin/sh", "-c", "env | grep APP" ]
  env:
    - name: APPLOGLEVEL          # 定义环境变量名称
      valueFrom:                  # key“apploglevel”对
应的值
        configMapKeyRef:
          name: cm-appvars        # 环境变量的值取自cm-
appvars中:
            key: apploglevel      # key为“apploglevel”
    - name: APPDATADIR          # 定义环境变量名称
      valueFrom:                  # key“appdatadir”对应
的值
        configMapKeyRef:
          name: cm-appvars        # 环境变量的值取自cm-
appvars中:
            key: appdatadir       # key为“appdatadir”
  restartPolicy: Never

```

使用kubect1 create-f命令创建该Pod，由于是测试Pod，所以该Pod在执行完启动命令后将会退出，并且不会被系统自动重启（restartPolicy=Never）：

```

# kubect1 create -f cm-test-pod.yaml
pod "cm-test-pod" created

```

使用`kubectl get pods--show-all`查看已经停止的Pod:

```
# kubectl get pods --show-all
```

	NAME	READY	STATUS	RESTARTS
AGE				
	cm-test-pod	0/1	Completed	0
8s				

查看该Pod的日志，可以看到启动命令“`env|grep APP`”的执行结果如下:

```
# kubectl logs cm-test-pod
```

```
APPDATADIR=/var/data
```

```
APPLEVEL=info
```

说明容器内部的环境变量使用ConfigMap `cm-appvars`中的值进行了正确的设置。

5.ConfigMap的使用: volumeMount模式

下面`cm-appconfigfiles.yaml`的例子中包含两个配置文件的定义:`server.xml`和`logging.properties`。

```
cm-appconfigfiles.yaml
```

```
apiVersion: v1
```

```
kind: ConfigMap
```

```
metadata:
```



```

    name: cm-serverxml
data:
key-serverxml: |
< xml version='1.0' encoding='utf-8' >
<Server port="8005" shutdown="SHUTDOWN">
<Listener
className="org.apache.catalina.startup.VersionLoggerListene
r" />
<Listener
className="org.apache.catalina.core.AprLifecycleListener"
SSLEngine="on" />
<Listener
className="org.apache.catalina.core.JreMemoryLeakPrevention
Listener" />
<Listener
className="org.apache.catalina.mbeans.GlobalResourcesLifecy
cleListener" />
<Listener
className="org.apache.catalina.core.ThreadLocalLeakPreventi
onListener" />
    <GlobalNamingResources>
    <Resource name="UserDatabase" auth="Container"

type="org.apache.catalina.UserDatabase"
description="User database that can
be updated and saved"

```

```

factory="org.apache.catalina.users.MemoryUserDatabaseFactor
y"

        pathname="conf/tomcat-users.xml" />
    </GlobalNamingResources>

    <Service name="Catalina">
        <Connector port="8080" protocol="HTTP/1.1"
            connectionTimeout="20000"
            redirectPort="8443" />
        <Connector port="8009" protocol="AJP/1.3"
redirectPort="8443" />
        <Engine name="Catalina" defaultHost="localhost">

            <Realm
className="org.apache.catalina.realm.LockOutRealm">

            <Realm
className="org.apache.catalina.realm.UserDatabaseRealm"
            resourceName="UserDatabase"/>
        </Realm>
        <Host name="localhost" appBase="webapps"
            unpackWARs="true" autoDeploy="true">

            <Valve
className="org.apache.catalina.valves.AccessLogValve"
            directory="logs"

                prefix="localhost_access_log"

            suffix=".txt"

                pattern="%h %l %u %t &quot;%r&quot;
%s %b" />

```

```

</Host>
</Engine>
</Service>
</Server>
key-loggingproperties: "handlers
                        = 1catalina.org.apache.juli.FileHandler,
2localhost.org.apache.juli.FileHandler,
                        3manager.org.apache.juli.FileHandler, 4host-
manager.org.apache.juli.FileHandler,
                        java.util.logging.ConsoleHandler\r\n\r\n.handlers
= 1catalina.org.apache.juli.FileHandler,

java.util.logging.ConsoleHandler\r\n\r\n1catalina.org.apach
e.juli.FileHandler.level
                        =
FINE\r\n1catalina.org.apache.juli.FileHandler.directory      =
${catalina.base}/logs\r\n1catalina.org.apache.juli.FileHand
ler.prefix
                        =
catalina.\r\n\r\n2localhost.org.apache.juli.FileHandler.lev
el
                        =
FINE\r\n2localhost.org.apache.juli.FileHandler.directory
                        =
${catalina.base}/logs\r\n2localhost.org.apache.juli.FileHan
dler.prefix
                        =
localhost.\r\n\r\n3manager.org.apache.juli.FileHandler.leve

```

1

```
=
FINE\r\n3manager.org.apache.juli.FileHandler.directory      =
${catalina.base}/logs\r\n3manager.org.apache.juli.FileHandl
er.prefix
                                =    manager.\r\n\r\n4host-
manager.org.apache.juli.FileHandler.level  =  FINE\r\n4host-
manager.org.apache.juli.FileHandler.directory
                                =    ${catalina.base}/logs\r\n4host-
manager.org.apache.juli.FileHandler.prefix =
                                host-
manager.\r\n\r\n4java.util.logging.ConsoleHandler.level    =
FINE\r\n4java.util.logging.ConsoleHandler.formatter
                                =
java.util.logging.SimpleFormatter\r\n\r\n\r\n4norg.apache.cat
alina.core.ContainerBase.[Catalina].[localhost].level
                                =  INFO\r\n4norg.apache.catalina.core.ContainerBase.
[Catalina].[localhost].handlers
                                =
2localhost.org.apache.juli.FileHandler\r\n\r\n4norg.apache.ca
talina.core.ContainerBase.[Catalina].[localhost].
[/manager].level
                                =  INFO\r\n4norg.apache.catalina.core.ContainerBase.
[Catalina].[localhost].[/manager].handlers
                                =
3manager.org.apache.juli.FileHandler\r\n\r\n4norg.apache.cata
lina.core.ContainerBase.[Catalina].[localhost].[/host-
```

```
manager].level
    = INFO\r\norg.apache.catalina.core.ContainerBase.
[Catalina].[localhost].[/host-manager].handlers
                                =      4host-
manager.org.apache.juli.FileHandler\r\n\r\n"
```

在Pod“cm-test-app”的定义中，将ConfigMap“cm-appconfigfiles”中的内容以文件的形式mount到容器内部的/configfiles目录中去。Pod配置文件cm-test-app.yaml的内容如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: cm-test-app
spec:
  containers:
  - name: cm-test-app
    image: kubeguide/tomcat-app:v1
    ports:
    - containerPort: 8080
    volumeMounts:
    - name: serverxml          # 引用volume名
      mountPath: /configfiles  # 挂载到容器内的目录
  volumes:
  - name: serverxml          # 定义volume名
    configMap:
      name: cm-appconfigfiles # 使用
```

ConfigMap“cm-appconfigfiles”

items:

```
- key: key-serverxml          # key=key-serverxml
  path: server.xml            # value将server.xml
```

文件名进行挂载

```
- key: key-loggingproperties  # key=key-loggingproperties
```

```
  path: logging.properties    # value将logging.properties文件名进行挂载
```

创建该Pod:

```
# kubectl create -f cm-test-app.yaml
```

```
pod "cm-test-app" created
```

登录容器，查看到 /configfiles 目录下存在 server.xml 和 logging.properties 文件，它们的内容就是 ConfigMap“cm-appconfigfiles”中两个key定义的内容。

```
# kubectl exec -ti cm-test-app -- bash
```

```
root@cm-test-app:/# cat /configfiles/server.xml
```

```
< xml version='1.0' encoding='utf-8' >
```

```
<Server port="8005" shutdown="SHUTDOWN">
```

```
.....
```

```
root@cm-test-app:/# cat
```

```
/configfiles/logging.properties
```

```
handlers = 1catalina.org.apache.juli.AsyncFileHandler,
2localhost.org.apache.juli.AsyncFileHandler,
3manager.org.apache.juli.AsyncFileHandler,          4host-
manager.org.apache.juli.AsyncFileHandler,
java.util.logging.ConsoleHandler
.....
```

如果在引用ConfigMap时不指定items，则使用volumeMount方式在容器内的目录中为每个item生成一个文件名为key的文件。

Pod配置文件cm-test-app.yaml的内容如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: cm-test-app
spec:
  containers:
  - name: cm-test-app
    image: kubeguide/tomcat-app:v1
    imagePullPolicy: Never
    ports:
    - containerPort: 8080
    volumeMounts:
    - name: serverxml          # 引用volume名
      mountPath: /configfiles # 挂载到容器内的目录
  volumes:
```

```
- name: serverxml                # 定义volume名
  configMap:
    name: cm-appconfigfiles      # 使用
    ConfigMap "cm-appconfigfiles"
```

创建该Pod:

```
# kubectl create -f cm-test-app.yaml
pod "cm-test-app" created
```

登录容器，查看到/configfiles目录下存在key-loggingproperties和key-serverxml文件，文件的名称来自ConfigMap cm-appconfigfiles中定义的两个key的名称，文件的内容则为value的内容:

```
# ls /configfiles
key-loggingproperties  key-serverxml
```

6.使用ConfigMap的限制条件

使用ConfigMap的限制条件如下。

- ConfigMap必须在Pod之前创建。
- ConfigMap也可以定义为属于某个Namespace。只有处于相同Namespaces中的Pod可以引用它。
- ConfigMap中的配额管理还未能实现。
- kubelet只支持可以被APIServer管理的Pod使用ConfigMap。kubelet在本Node上通过--manifest-url或--config自动创建的静态Pod将无法

引用ConfigMap。

- 在Pod对ConfigMap进行挂载（volumeMount）操作时，容器内部只能挂载为“目录”，无法挂载为“文件”。在挂载到容器内部后，目录中将包含ConfigMap定义的每个item，如果该目录下原先还有其他文件，则容器内的该目录将会被挂载的ConfigMap进行覆盖。如果应用程序需要保留原来的其他文件，则需要进行额外的处理。可以通过将ConfigMap挂载到容器内部的临时目录，再通过启动脚本将配置文件复制或者链接（cp或link操作）到应用所用的实际配置目录下。

2.4.6 Pod生命周期和重启策略

Pod在整个生命周期过程中被系统定义为各种状态，熟悉Pod的各种状态对于我们理解如何设置Pod的调度策略、重启策略是很有必要的。

Pod的状态包括以下几种，如表2.14所示。

表2.14 Pod的状态

状态值	描述
Pending	API Server 已经创建该 Pod，但 Pod 内还有一个或多个容器的镜像没有创建，包括正在下载镜像的过程
Running	Pod 内所有容器均已创建，且至少有一个容器处于运行状态、正在启动状态或正在重启状态
Succeeded	Pod 内所有容器均成功执行退出，且不会再重启
Failed	Pod 内所有容器均已退出，但至少有一个容器退出为失败状态
Unknown	由于某种原因无法获取该 Pod 的状态，可能由于网络通信不畅导致

Pod的重启策略（RestartPolicy）应用于Pod内的所有容器，并且仅在Pod所处的Node上由kubelet进行判断和重启操作。当某个容器异常退出或者健康检查（详见下节）失败时，kubelet将根据RestartPolicy的设置来进行相应的操作。

Pod的重启策略包括 Always、OnFailure 和 Never，默认值为 Always。

- Always: 当容器失效时，由kubelet自动重启该容器。
- OnFailure: 当容器终止运行且退出码不为0时，由kubelet自动重启该容器。

- **Never**: 不论容器运行状态如何，**kubelet**都不会重启该容器。

kubelet重启失效容器的时间间隔以**sync-frequency**乘以2n来计算，例如1、2、4、8倍等，最长延时5分钟，并且在成功重启后的10分钟后重置该时间。

Pod的重启策略与控制方式息息相关，当前可用于管理**Pod**的控制器包括**ReplicationController**、**Job**、**DaemonSet**及直接通过**kubelet**管理（静态**Pod**）。每种控制器对**Pod**的重启策略要求如下。

- **RC**和**DaemonSet**: 必须设置为**Always**，需要保证该容器持续运行。
- **Job**: **OnFailure**或**Never**，确保容器执行完成后不再重启。
- **kubelet**: 在**Pod**失效时自动重启它，不论**RestartPolicy**设置为什么值，并且也不会对**Pod**进行健康检查。

结合**Pod**的状态和重启策略，表2.15列出一些常见的状态转换场景。

表2.15 一些常见的状态转换场景

Pod 包含的容器数	Pod 当前的状态	发 生 事 件	Pod 的结果状态		
			RestartPolicy= Always	RestartPolicy= OnFailure	RestartPolicy= Never
包含 1 个容器	Running	容器成功退出	Running	Succeeded	Succeeded
包含 1 个容器	Running	容器失败退出	Running	Running	Failed
包含两个容器	Running	1 个容器失败退出	Running	Running	Running
包含两个容器	Running	容器被 OOM 杀掉	Running	Running	Failed

2.4.7 Pod健康检查

对Pod的健康状态检查可以通过两类探针来检查：LivenessProbe和ReadinessProbe。

- **LivenessProbe探针**：用于判断容器是否存活（running状态），如果LivenessProbe探针探测到容器不健康，则kubelet将杀掉该容器，并根据容器的重启策略做相应的处理。如果一个容器不包含LivenessProbe探针，那么kubelet认为该容器的LivenessProbe探针返回的值永远是“Success”。
- **ReadinessProbe**：用于判断容器是否启动完成（ready状态），可以接收请求。如果ReadinessProbe探针检测到失败，则Pod的状态将被修改。Endpoint Controller将从Service的Endpoint中删除包含该容器所在Pod的Endpoint。

kubelet定期执行LivenessProbe探针来诊断容器的健康状况。LivenessProbe有以下三种实现方式。

（1）ExecAction：在容器内部执行一个命令，如果该命令的返回码为0，则表明容器健康。

在下面的例子中，通过执行“cat/tmp/health”命令来判断一个容器运行是否正常。而该Pod运行之后，在创建/tmp/health文件的10秒之后将删除该文件，而LivenessProbe健康检查的初始探测时间（initialDelaySeconds）为15秒，探测结果将是Fail，将导致kubelet杀掉该容器并重启它。

```
  apiVersion: v1
  kind: Pod
  metadata:
    labels:
      test: liveness
      name: liveness-exec
  spec:
    containers:
      - name: liveness
        image: gcr.io/google_containers/busybox
        args:
          - /bin/sh
          - -c
            - echo ok > /tmp/health; sleep 10; rm -rf
/tmp/health; sleep 600
        livenessProbe:
          exec:
            command:
              - cat
              - /tmp/health
            initialDelaySeconds: 15
            timeoutSeconds: 1
```

(2) **TCPSocketAction**: 通过容器的IP地址和端口号执行TCP检查，如果能够建立TCP连接，则表明容器健康。

在下面的例子中，通过与容器内的localhost: 80建立TCP连接进行健康检查。

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-healthcheck
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    livenessProbe:
      tcpSocket:
        port: 80
      initialDelaySeconds: 30
      timeoutSeconds: 1
```

(3) HTTPGetAction: 通过容器的IP地址、端口号及路径调用HTTP Get方法，如果响应的状态码大于等于200且小于等于400，则认为容器状态健康。

在下面的例子中， kubelet 定时发送HTTP请求到localhost: 80/_status/healthz来进行容器应用的健康检查。

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-healthcheck
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    livenessProbe:
      httpGet:
        path: /_status/healthz
        port: 80
      initialDelaySeconds: 30
      timeoutSeconds: 1
```

对于每种探测方式，都需要设置 `initialDelaySeconds` 和 `timeoutSeconds` 两个参数，它们的含义分别如下。

- **initialDelaySeconds**: 启动容器后进行首次健康检查的等待时间，单位为秒。
- **timeoutSeconds**: 健康检查发送请求后等待响应的超时时间，单位为秒。当超时发生时，`kubelet` 会认为容器已经无法提供服务，将会重启该容器。

2.4.8 玩转Pod调度

在Kubernetes系统中，Pod在大部分场景下都只是容器的载体而已，通常需要通过RC、Deployment、DaemonSet、Job等对象来完成Pod的调度与自动控制功能。

1.RC、Deployment：全自动调度

RC的主要功能之一就是自动部署一个容器应用的多份副本，以及持续监控副本的数量，在集群内始终维持用户指定的副本数量。

根据 frontend-controller.yaml 配置，用户需要创建3个 kubeguide/guestbook-php-frontend 应用的副本，在将该定义发送给 Kubernetes 之后，系统将在集群中合适的Node上创建3个Pod，并始终维持3个副本的数量。

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  replicas: 3
  selector:
```



```
    name: frontend
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
        - name: frontend
          image: kubeguide/guestbook-php-frontend
          env:
            - name: GET_HOSTS_FROM
              value: env
          ports:
            - containerPort: 80
```

在调度策略上，除了使用系统内置的调度算法选择合适的Node进行调度，也可以在Pod的定义中使用NodeSelector或NodeAffinity来指定满足条件的Node进行调度，下面我们分别进行说明。

1) NodeSelector: 定向调度

Kubernetes Master上的Scheduler服务（kube-scheduler进程）负责实现Pod的调度，整个调度过程通过执行一系列复杂的算法，最终为每个Pod计算出一个最佳的目标节点，这一过程是自动完成的，通常我们无法知道Pod最终会被调度到哪个节点上。在实际情况中，也可能需要将Pod调度到指定的一些Node上，可以通过Node的标签（Label）和Pod的nodeSelector属性相匹配，来达到上述目的。

(1) 首先通过`kubectl label`命令给目标Node打上一些标签:

```
kubectl label nodes <node-name><label-key>=<label-value>
```

这里, 我们为`k8s-node-1`节点打上一个`zone=north`的标签, 表明它是“北方”的一个节点:

```
$ kubectl label nodes k8s-node-1 zone=north
```

	NAME		LABELS
STATUS			
	k8s-node-1		kubernetes.io/hostname=k8s-node-1, zone=north
	Ready		

上述命令行操作也可以通过修改资源定义文件的方式, 并执行`kubectl replace-f xxx.yaml`命令来完成。

(2) 然后, 在Pod的定义中加上`nodeSelector`的设置, 以`redis-master-controller.yaml`为例:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-master
  labels:
    name: redis-master
spec:
  replicas: 1
```

```
selector:
  name: redis-master
template:
  metadata:
    labels:
      name: redis-master
  spec:
    containers:
      - name: master
        image: kubeguide/redis-master
        ports:
          - containerPort: 6379
    nodeSelector:
      zone: north
```

运行`kubectl create-f`命令创建Pod，scheduler就会将该Pod调度到拥有`zone=north`标签的Node上。

使用`kubectl get pods-o wide`命令可以验证Pod所在的Node:

```
# kubectl get pods -o wide
```

	NAME	READY	STATUS	RESTARTS
AGE	NODE			
	redis-master-f0rqj	1/1	Running	0
19s	k8s-node-1			

如果我们给多个Node都定义了相同的标签（例如`zone=north`），则scheduler将会根据调度算法从这组Node中挑选一个可用的Node进行Pod调度。

通过基于Node标签的调度方式，我们可以把集群中具有不同特点的Node贴上不同的标签，例如“`role=frontend`”“`role=backend`”“`role=database`”等标签，在部署应用时就可以根据应用的需求设置NodeSelector来进行指定Node范围的调度。

需要注意的是，如果我们指定了Pod的nodeSelector条件，且集群中不存在包含相应标签的Node，则即使集群中还有其他可供使用的Node，这个Pod也无法被成功调度。

2) NodeAffinity: 亲和性调度

NodeAffinity意为Node亲和性的调度策略，是将来替换NodeSelector的下一代调度策略。由于NodeSelector通过Node的Label进行精确匹配，所以NodeAffinity增加了In、NotIn、Exists、DoesNotExist、Gt、Lt等操作符来选择Node，能够使调度策略更加灵活。同时，在NodeAffinity中将增加一些信息来设置亲和性调度策略。

- RequiredDuringSchedulingRequiredDuringExecution：类似于NodeSelector，但在Node不满足条件时，系统将从该Node上移除之前调度上的Pod。
- RequiredDuringSchedulingIgnoredDuringExecution：与第1个RequiredDuringSchedulingRequiredDuringExecution的作用相似，

区别是在Node不满足条件时，系统不一定从该Node上移除之前调度上的Pod。

- **PreferredDuringSchedulingIgnoredDuringExecution**：指定在满足调度条件的Node中，哪些Node应更优先地进行调度。同时在Node不满足条件时，系统不一定从该Node上移除之前调度上的Pod。

在当前的Alpha版本中，需要在Pod的metadata.annotations中设置NodeAffinity的内容：

```
apiVersion: v1
kind: Pod
metadata:
  name: with-labels
  annotations:
    scheduler.alpha.kubernetes.io/affinity: >
    {
      "nodeAffinity": {
        "requiredDuringSchedulingIgnoredDuringExecution": {
          "nodeSelectorTerms": [
            {
              "matchExpressions": [
                {
                  "key": "kubernetes.io/e2e-az-
name",
                  "operator": "In",
                  "values": ["e2e-az1", "e2e-az2"]
```

```

        }
    ]
}
]
}
}
}
    }
    another-annotation-key: another-annotation-value
spec:
  containers:
  - name: with-labels
    image: gcr.io/google_containers/pause:2.0

```

这里 **NodeAffinity** 的设置说明只有 **Node** 的 **Label** 中包含 **key=kubernetes.io/e2e-az-name**，并且其 **value** 为 “e2e-az1” 或 “e2e-az2” 时，才能成为该 **Pod** 的调度目标。其中操作符为 **In**，代表“或”运算，其他操作符包括 **NotIn**（不属于）、**Exists**（存在一个条件）、**DoesNotExist**（不存在）、**Gt**（大于）、**Lt**（小于）。

如果同时设置了 **NodeSelector** 和 **NodeAffinity**，则系统将需要同时满足两者的设置才能进行调度。

在未来的 **Kubernetes** 版本中，还将加入 **Pod Affinity** 的设置，用于控制当调度 **Pod** 到某个特定的 **Node** 上时，判断是否有其他 **Pod** 正在该 **Node** 上运行，即通过其他的相关 **Pod** 进行调度，而不仅仅通过 **Node** 本身的标签进行调度。

2.DaemonSet: 特定场景调度

DaemonSet是Kubernetes 1.2版本新增的一种资源对象，用于管理在集群中每个Node上仅运行一份Pod的副本实例，如图2.10所示。

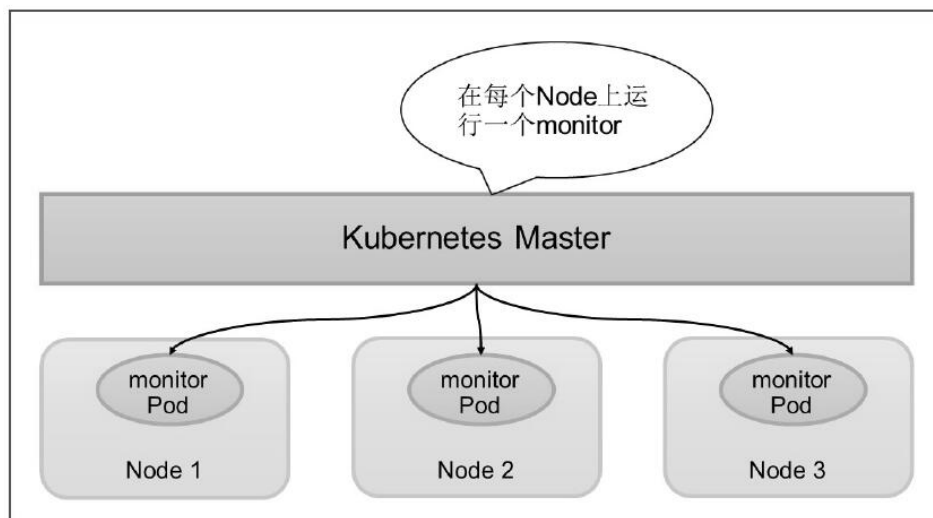


图2.10 DaemonSet示例

这种用法适合一些有这种需求的应用。

- 在每个Node上运行一个GlusterFS存储或者Ceph存储的daemon进程。
- 在每个Node上运行一个日志采集程序，例如fluentd或者logstash。
- 在每个Node上运行一个健康程序，采集该Node的运行性能数据，例如Prometheus Node Exporter、collectd、New Relic agent或者Ganglia gmond等。

DaemonSet的Pod调度策略与RC类似，除了使用系统内置的算法在每台Node上进行调度，也可以在Pod的定义中使用NodeSelector或NodeAffinity来指定满足条件的Node范围进行调度。

下面的例子定义为在每台Node上启动一个fluentd容器，配置文件fluentd-ds.yaml的内容如下，其中挂载了物理机的两个目录“/var/log”和“/var/lib/docker/containers”：

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: fluentd-cloud-logging
  namespace: kube-system
  labels:
    k8s-app: fluentd-cloud-logging
spec:
  template:
    metadata:
      namespace: kube-system
      labels:
        k8s-app: fluentd-cloud-logging
    spec:
      containers:
        - name: fluentd-cloud-logging
          image: gcr.io/google_containers/fluentd-
elasticsearch:1.17
          resources:
            limits:
              cpu: 100m
              memory: 200Mi
      env:
```



```
- name: FLUENTD_ARGS
  value: -q
volumeMounts:
- name: varlog
  mountPath: /var/log
  readOnly: false
- name: containers
  mountPath: /var/lib/docker/containers
  readOnly: false
volumes:
- name: containers
  hostPath:
    path: /var/lib/docker/containers
- name: varlog
  hostPath:
    path: /var/log
```

使用**kubectl create**命令创建该**DaemonSet**:

```
# kubectl create -f fluentd-ds.yaml
daemonset "fluentd-cloud-logging" created
```

查看创建好的**DaemonSet**和**Pod**，可以看到在每个**Node**上都创建了一个**Pod**:

```
# kubectl get daemonset --namespace=kube-system
```

NAME	DESIRED	CURRENT	NODE-
------	---------	---------	-------

```

SELECTOR    AGE
    fluentd-cloud-logging    2    2    <none>
3s

# kubectl get pods --namespace=kube-system
NAME                                READY    STATUS
RESTARTS    AGE
    fluentd-cloud-logging-7tw9z    1/1    Running    0
1h
    fluentd-cloud-logging-aqdn1    1/1    Running    0
1h

```

3.Job: 批处理调度

Kubernetes从1.2版本开始支持批处理类型的应用，我们可以通过KubernetesJob资源对象来定义并启动一个批处理任务。批处理任务通常并行（或者串行）启动多个计算进程去处理一批工作项（**work item**），处理完成后，整个批处理任务结束。按照批处理任务实现方式的不同，批处理任务可以分为如图2.11所示的几种模式。

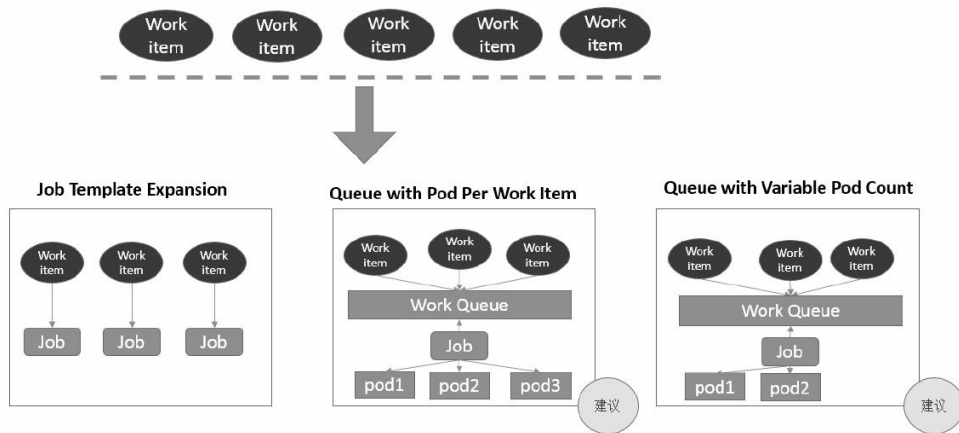


图2.11 批处理任务的几种模式

- **Job Template Expansion模式：** 一个Job对象对应一个待处理的Work item，有几个Work item就产生几个独立的Job，通常适合Work item数量少、每个Work item要处理的数据量比较大的场景，比如有一个100GB的文件作为一个Work item，总共10个文件需要处理。
- **Queue with Pod Per Work Item模式：** 采用一个任务队列存放Work item，一个Job对象作为消费者去完成这些Work item，在这种模式下，Job会启动N个Pod，每个Pod对应一个Work item。
- **Queue with Variable Pod Count模式：** 也是采用一个任务队列存放Work item，一个Job对象作为消费者去完成这些Work item，但与上面的模式不同，Job启动的Pod数量是可变的。

还有一种被称为Single Job with Static Work Assignment的模式，也是一个Job产生多个Pod的模式，但它采用程序静态方式分配任务项，而不是采用队列模式进行动态分配。

如表2.16所示是这几种模式的一个对比。

模式名称	是否是一个Job	Pod 的数量少于 Work item	用户程序是否要做相应的修改	Kubernetes 是否支持
Job Template Expansion	/	/	是	是
Queue with Pod Per Work Item	是	/	有时候需要	是
Queue with Variable Pod Count	是	/	/	是
Single Job with Static Work Assignment	是	/	是	/

考虑到批处理的并行问题，Kubernetes将Job分以下三种类型。

1) Non-parallel Jobs

通常一个Job只启动一个Pod，则除非Pod异常，才会重启该Pod，一旦此Pod正常结束，Job将结束。

2) Parallel Jobs with a fixed completion count

并行Job会启动多个Pod，此时需要设定Job的.spec.completions参数为一个正数，当正常结束的Pod数量达到此参数设定的值后，Job结束。此外，Job的.spec.parallelism参数用来控制并行度，即同时启动几个Job来处理Work Item。

3) Parallel Jobs with a work queue

任务队列方式的并行Job需要一个独立的Queue，Work item都在一个Queue中存放，不能设置Job的.spec.completions参数，此时Job有以下一些特性。

- 每个Pod能独立判断和决定是否还有任务项需要处理。
- 如果某个Pod正常结束，则Job不会再启动新的Pod。
- 如果一个Pod成功结束，则此时应该不存在其他Pod还在干活的情况，它们应该都处于即将结束、退出的状态。

- 如果所有Pod都结束了，且至少有一个Pod成功结束，则整个Job算是成功结束。

下面我们分别说说常见的三种批处理模型在Kubernetes中的例子。

首先是Job Template Expansion模式，由于这种模式下每个Work item对应一个Job实例，所以这种模式首先定义一个Job模板，模板里主要的参数是Work item的标识，因为每个Job处理不同的Work item。如下所示的Job模板（文件名为job.yaml.txt）中的\$ITEM可以作为任务项的标识：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: process-item-$ITEM
  labels:
    jobgroup: jobexample
spec:
  template:
    metadata:
      name: jobexample
      labels:
        jobgroup: jobexample
    spec:
      containers:
        - name: c
```

```
        image: busybox
        command: ["sh", "-c", "echo Processing item
$ITEM&& sleep 5"]
        restartPolicy: Never
```

通过下面的操作，生成3个对应的Job定义文件并创建Job:

```
# for i in apple banana cherry
> do
    > cat job.yaml.txt | sed "s/\$ITEM/\$i/" >
./jobs/job-\$i.yaml
> done
# ls jobs
job-apple.yaml  job-banana.yaml  job-cherry.yaml
# kubectl create -f jobs
job "process-item-apple" created
job "process-item-banana" created
job "process-item-cherry" created
```

观察Job的运行情况:

```
# kubectl get jobs -l jobgroup=jobexample
```

NAME	DESIRED	SUCCESSFUL	AGE
process-item-apple	1	1	4m
process-item-banana	1	1	4m
process-item-cherry	1	1	4m

其次，我们看看Queue with Pod Per Work Item模式，在这种模式下需要一个任务队列存放Work item，比如RabbitMQ，客户端程序先把要处理的任務变成Work item放入到任务队列，然后编写Worker程序并打包镜像并定义成为Job中的Work Pod，Worker程序的实现逻辑是从任务队列中拉取一个Work item并处理，处理完成后即结束进程，图2.12给出了并行度为2的一个Demo示意图。

最后，我们再看看Queue with Variable Pod Count模式，如图2.13所示，由于这种模式下，Worker程序需要知道队列中是否还有等待处理的Work item，如果有就取出来并处理，否则就认为所有工作完成并结束进程，所以任务队列通常要采用Redis或者数据库来实现。

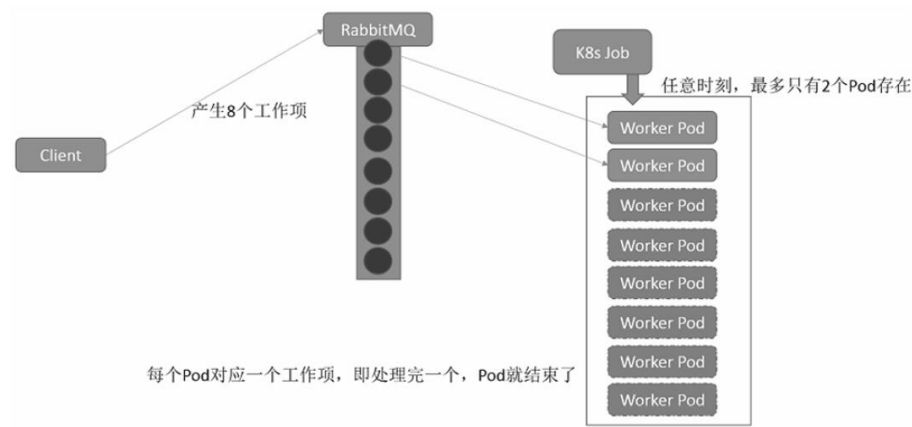


图2.12 Queue with Pod Per Work Item示例

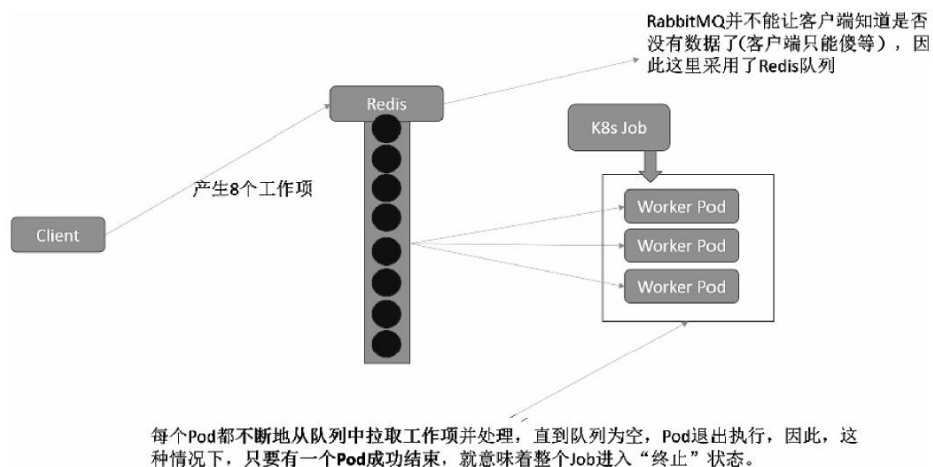


图2.13 Queue with Variable Pod Count示例

Kubernetes对Job的支持还处于初级阶段，类似Linux Cron的定时任务也还没时间，计划在Kubernetes 1.4中实现。此外，更为复杂的流程类的批处理框架也还没有考虑，但随着Kubernetes生态圈的不断发展和壮大，相信Kubernetes在批处理方面也会有更多的规划。

2.4.9 Pod的扩容和缩容

在实际生产系统中，我们经常会遇到某个服务需要扩容的场景，也可能会遇到由于资源紧张或者工作负载降低而需要减少服务实例数量的场景。此时我们可以利用RC的Scale机制来完成这些工作。以redis-slave RC为例，已定义的最初副本数量为2，通过kubect scale命令可以将redis-slave RC控制的Pod副本数量从初始的2更新为3：

```
$ kubectl scale rc redis-slave --replicas=3
replicationcontroller "redis-slave" scaled
```

执行kubectl get pods命令来验证Pod的副本数量增加到3：

```
$ kubectl get pods
```

	NAME	READY	STATUS	RESTARTS
AGE				
	redis-slave-4na2n	1/1	Running	0
	redis-slave-92u3k	1/1	Running	0
	redis-slave-palab	1/1	Running	0

将--replicas设置为比当前Pod副本数量更小的数字，系统将会“杀掉”一些运行中的Pod，以实现应用集群缩容：

```
$ kubectl scale rc redis-slave --replicas=1
replicationcontroller "redis-slave" scaled
```

```
$ kubectl get pods
```

NAME	READY	STATUS
RESTARTS AGE		
redis-slave-4na2n	1/1	Running
1h		0

除了可以手工通过**kubectl scale**命令完成Pod的扩容和缩容操作，Kubernetes v1.1版本新增了名为**Horizontal Pod Autoscaler (HPA)**的控制器，用于实现基于CPU使用率进行自动Pod扩缩容的功能。HPA控制器基于Master的**kube-controller-manager**服务启动参数**--horizontal-pod-autoscaler-sync-period**定义的时长（默认为30秒），周期性地监测目标Pod的CPU使用率，并在满足条件时对**ReplicationController**或**Deployment**中的Pod副本数量进行调整，以符合用户定义的平均Pod CPU使用率。Pod CPU使用率来源于**heapster**组件，所以需要预先安装好**heapster**。

创建HPA时可以使用**kubectl autoscale**命令进行快速创建或者使用yaml配置文件进行创建。在创建HPA之前，需要已经存在一个RC或Deployment对象，并且该RC或Deployment中的Pod必须定义**resources.requests.cpu**的资源请求值，如果不设置该值，则**heapster**将无法采集到该Pod的CPU使用情况，会导致HPA无法正常工作。

下面通过给一个RC设置HPA，然后使用一个客户端对其进行压力测试，对HPA的用法进行示例。

以php-apache的RC为例，设置cpu request为200m，未设置limit上限的值：

```
php-apache-rc.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: php-apache
spec:
  replicas: 1
  template:
    metadata:
      name: php-apache
      labels:
        app: php-apache
    spec:
      containers:
      - name: php-apache
        image: gcr.io/google_containers/hpa-example
        resources:
          requests:
            cpu: 200m
        ports:
        - containerPort: 80

# kubectl create -f php-apache-rc.yaml
replicationcontroller "php-apache" created
```

再创建一个php-apache的Service，供客户端访问：

```
php-apache-svc.yaml
apiVersion: v1
kind: Service
metadata:
  name: php-apache
spec:
  ports:
    - port: 80
  selector:
    app: php-apache

# kubectl create -f php-apache-svc.yaml
service "php-apache" created
```

接下来为RC“php-apache”创建一个HPA控制器，在1和10之间调整Pod的副本数量，以使得平均Pod CPU使用率维持在50%。

使用kubectl autoscale命令进行创建：

```
# kubectl autoscale rc php-apache --min=1 --max=10 --
cpu-percent=50
```

或者通过yaml配置文件来创建HPA，需要在scaleTargetRef字段指定需要管理的RC或Deployment的名字，然后设置minReplicas、maxReplicas和targetCPUUtilizationPercentage参数：

```
hpa-php-apache.yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: v1
    kind: ReplicationController
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50

# kubectl create -f hpa-php-apache.yaml
horizontalpodautoscaler "php-apache" created
```

查看已创建的HPA:

```
# kubectl get hpa
```

	NAME	REFERENCE	TARGET
CURRENT	MINPODS	MAXPODS	AGE
	php-apache	ReplicationController/php-apache	50%
0%	1	10	1m

然后，我们创建一个busybox Pod，用于对php-apache服务发起压力测试的请求:

```
busybox-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: busybox
spec:
  containers:
  - name: busybox
    image: busybox
    command: [ "sleep", "3600" ]

# kubectl create -f busybox-pod.yaml
pod "busybox" created
```

登录busybox容器，执行一个无限循环的wget命令来访问php-apache服务：

```
# while true; do wget -q -O- http://php-apache >
/dev/null; done
```

注意这里wget的目的地URL地址是Service的名称“php-apache”，这要求DNS服务正常工作，也可以使用Service的虚拟ClusterIP地址对其进行访问，例如http://169.169.122.145：

```
# kubectl exec -ti busybox -- sh
/ # while true; do wget -q -O- http://php-apache >
/dev/null; done
```

等待一段时间后，观察HPA控制器搜集到的Pod CPU使用率：

```
# kubectl get hpa
```

NAME		REFERENCE		TARGET
CURRENT	MINPODS	MAXPODS	AGE	
3068%	1	10	3m	php-apache
				ReplicationController/php-apache
				50%

再过一会儿，查看RC php-apache副本数量的变化：

```
# kubectl get rc
```

NAME	DESIRED	CURRENT	AGE
php-apache	10	10	23m

可以看到HPA已经根据Pod的CPU使用率的提高对RC进行了自动扩容，Pod副本数量变成了10个。这个过程如图2.14所示。

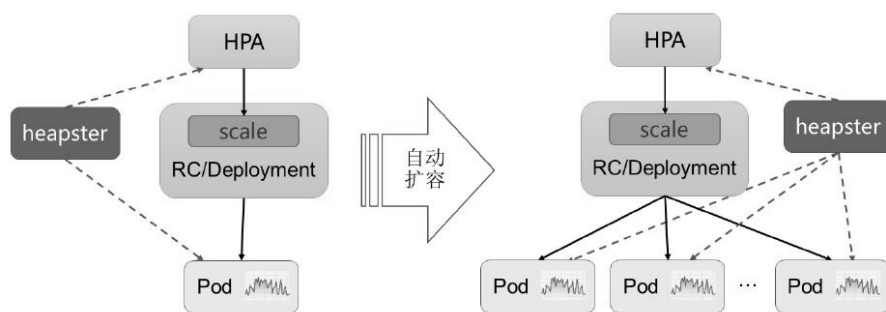


图2.14 HPA自动扩容

最后，我们停止压力测试，在busybox的控制台输入Ctrl+C，停止无限循环操作。

等待一段时间，观察HPA的变化：

# kubectl get hpa					
NAME			REFERENCE		TARGET
CURRENT	MINPODS	MAXPODS	AGE		
	php-apache		ReplicationController/php-apache		50%
3%	1	10	20m		

再次查看RC的副本数量：

NAME	DESIRED	CURRENT	AGE
php-apache	1	1	26m

可以看到HPA根据Pod CPU使用率的降低对副本数量进行了缩容操作，Pod副本数量变成了1个。

当前HPA还只支持将CPU使用率作为Pod副本扩容缩容的触发条件，在将来的版本中，将会支持应用相关的指标例如QPS（queries per second）或平均响应时间作为触发条件。

2.4.10 Pod的滚动升级

下面我们说说Pod的升级问题。

当集群中的某个服务需要升级时，我们需要停止目前与该服务相关的所有Pod，然后重新拉取镜像并启动。如果集群规模比较大，则这个工作就变成了一个挑战，而且先全部停止然后逐步升级的方式会导致较长时间的服务不可用。Kubernetes提供了rolling-update（滚动升级）功能来解决上述问题。

滚动升级通过执行`kubectl rolling-update`命令一键完成，该命令创建了一个新的RC，然后自动控制旧的RC中的Pod副本的数量逐渐减少到0，同时新的RC中的Pod副本的数量从0逐步增加到目标值，最终实现了Pod的升级。需要注意的是，系统要求新的RC需要与旧的RC在相同的命名空间（Namespace）内，即不能把别人的资产偷偷转移到自家名下。滚动升级的过程如图2.15所示。

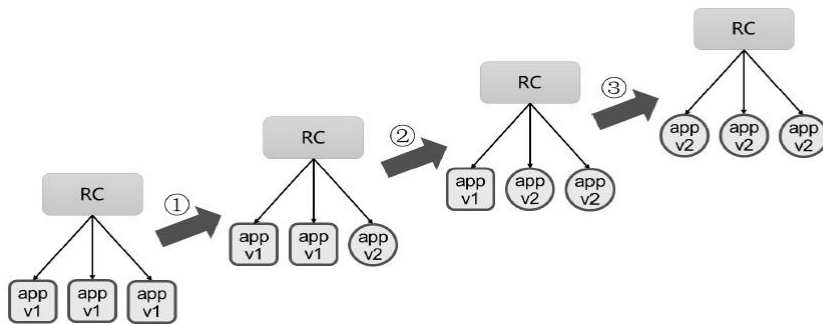


图2.15 Pod的滚动升级

以redis-master为例，假设当前运行的redis-master Pod是1.0版本，则现在需要升级到2.0版本。

创建redis-master-controller-v2.yaml的配置文件如下：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-master-v2
  labels:
    name: redis-master
    version: v2
spec:
  replicas: 1
  selector:
    name: redis-master
    version: v2
  template:
    metadata:
      labels:
        name: redis-master
        version: v2
    spec:
      containers:
        - name: master
```

```
image: kubeguide/redis-master:2.0
ports:
- containerPort: 6379
```

在配置文件中需要注意以下几点。

(1) RC的名字 (**name**) 不能与旧的RC的名字相同。

(2) 在**selector**中应至少有一个**Label**与旧的RC的**Label**不同，以标识其为新的RC。本例中新增了一个名为**version**的**Label**，以与旧的RC进行区分。

运行**kubectl rolling-update**命令完成Pod的滚动升级：

```
kubectl rolling-update redis-master -f redis-master-
controller-v2.yaml
```

kubectl的执行过程如下：

```
Creating redis-master-v2
At beginning of loop: redis-master replicas: 2, redis-
master-v2 replicas: 1
Updating redis-master replicas: 2, redis-master-v2
replicas: 1
At end of loop: redis-master replicas: 2, redis-
master-v2 replicas: 1
At beginning of loop: redis-master replicas: 1, redis-
master-v2 replicas: 2
```

```
    Updating redis-master replicas: 1, redis-master-v2
replicas: 2
    At end of loop: redis-master replicas: 1, redis-
master-v2 replicas: 2
    At beginning of loop: redis-master replicas: 0, redis-
master-v2 replicas: 3
    Updating redis-master replicas: 0, redis-master-v2
replicas: 3
    At end of loop: redis-master replicas: 0, redis-
master-v2 replicas: 3
    Update succeeded. Deleting redis-master
    redis-master-v2
```

等所有新的Pod启动完成后，旧的Pod也被全部销毁，这样就完成了容器集群的更新工作。

另一种方法是不使用配置文件，直接用kubectl rolling-update命令，加上--image参数指定新版镜像名称来完成Pod的滚动升级：

```
kubectl rolling-update redis-master --image=redis-
master:2.0
```

与使用配置文件的方式不同，执行的结果是旧的RC被删除，新的RC仍将使用旧的RC的名字。

kubectl的执行过程如下：

Creating redis-master-ea866a5d2c08588c3375b86fb253db75

At beginning of loop: redis-master replicas: 2, redis-master-ea866a5d2c08588c3375b86fb253db75 replicas: 1

Updating redis-master replicas: 2, redis-master-ea866a5d2c08588c3375b86fb253db75 replicas: 1

At end of loop: redis-master replicas: 2, redis-master-ea866a5d2c08588c3375b86fb253db75 replicas: 1

At beginning of loop: redis-master replicas: 1, redis-master-ea866a5d2c08588c3375b86fb253db75 replicas: 2

Updating redis-master replicas: 1, redis-master-ea866a5d2c08588c3375b86fb253db75 replicas: 2

At end of loop: redis-master replicas: 1, redis-master-ea866a5d2c08588c3375b86fb253db75 replicas: 2

At beginning of loop: redis-master replicas: 0, redis-master-ea866a5d2c08588c3375b86fb253db75 replicas: 3

Updating redis-master replicas: 0, redis-master-ea866a5d2c08588c3375b86fb253db75 replicas: 3

At end of loop: redis-master replicas: 0, redis-master-ea866a5d2c08588c3375b86fb253db75 replicas: 3

Update succeeded. Deleting old controller: redis-master

Renaming redis-master-ea866a5d2c08588c3375b86fb253db75 to redis-master

redis-master

可以看到，`kubectl`通过新建一个新版本Pod，停掉一个旧版本Pod，逐步迭代来完成整个RC的更新。

更新完成后，查看RC：

```
$ kubectl get rc
```

CONTROLLER	CONTAINER(S)	IMAGE(S)
redis-master	master	kubeguide/redis-master:2.0

```
deployment=ea866a5d2c08588c3375b86fb253db75,name=redis-master,version=v1 3
```

可以看到，`kubectl`给RC增加了一个key为“deployment”的Label（这个key的名字可通过`--deployment-label-key`参数进行修改），Label的值是RC的内容进行Hash计算后的值，相当于签名，这样就能很方便地比较RC里的Image名字及其他信息是否发生了变化，它的具体作用可以参见第6章的源码分析。

如果在更新过程中发现配置有误，则用户可以中断更新操作，并通过执行`kubectl rolling-update-rollback`完成Pod版本的回滚：

```
$ kubectl rolling-update redis-master --
image=kubeguide/redis-master:2.0 --rollback
```

```
Found existing update in progress (redis-master-
fef9752aa5883ca4d53013a7b583967), resuming.
```

```
Found desired replicas.Continuing update with existing
controller redis-master.
```

```
At beginning of loop: redis-master-  
fefd9752aa5883ca4d53013a7b583967 replicas: 0, redis-master  
replicas: 3
```

```
Updating redis-master-fefd9752aa5883ca4d53013a7b583967  
replicas: 0, redis-master replicas: 3
```

```
At end of loop: redis-master-  
fefd9752aa5883ca4d53013a7b583967 replicas: 0, redis-master  
replicas: 3
```

```
Update succeeded. Deleting redis-master-  
fefd9752aa5883ca4d53013a7b583967
```

```
redis-master
```

到此，可以看到Pod恢复到更新前的版本了。

2.5 深入掌握Service

Service是Kubernetes最核心的概念，通过创建Service，可以为一组具有相同功能的容器应用提供一个统一的入口地址，并且将请求进行负载分发到后端的各个容器应用上。本节对Service的使用进行详细说明，包括Service的负载均衡、外网访问、DNS服务的搭建、Ingress 7层路由机制等。

2.5.1 Service定义详解

yaml格式的Service定义文件的完整内容如下:

```
apiVersion: v1                                // Required
kind: Service                                  // Required
metadata:                                       // Required
  name: string                                 // Required
  namespace: string                            // Required
  labels:
    - name: string
  annotations:
    - name: string
spec:                                           // Required
  selector: []                                  // Required
  type: string                                  // Required
  clusterIP: string
  sessionAffinity: string
  ports:
    - name: string
      protocol: string
      port: int
      targetPort: int
      nodePort: int
```

```
status:
  loadBalancer:
    ingress:
      ip: string
      hostname: string
```

对各属性的说明如表2.17所示。

表2.17 对Service的定义文件模板的各属性的说明

属性名称	取值类型	是否必选	取值说明
version	string	Required	v1
kind	string	Required	Service
metadata	object	Required	元数据
metadata.name	string	Required	Service 名称，需符合 RFC 1035 规范
metadata.namespace	string	Required	命名空间，不指定系统时将使用名为“default”的命名空间
metadata.labels[]	list		自定义标签属性列表
metadata.annotation[]	list		自定义注解属性列表
spec	object	Required	详细描述
spec.selector[]	list	Required	Label Selector 配置，将选择具有指定 Label 标签的 Pod 作为管理范围
spec.type	string	Required	Service 的类型，指定 Service 的访问方式，默认为 ClusterIP。 ClusterIP: 虚拟的服务 IP 地址，该地址用于 Kubernetes 集群内部的 Pod 访问，在 Node 上 kube-proxy 通过设置的 iptables 规则进行转发。 NodePort: 使用宿主机的端口，使能够访问各 Node 的外部客户端通过 Node 的 IP 地址和端口号就能访问服务。 LoadBalancer: 使用外接负载均衡器完成到服务的负载分发，需要在 spec.status.loadBalancer 字段指定外部负载均衡器的 IP 地址，并同时定义 nodePort 和 clusterIP，用于公有云环境
spec.clusterIP	string		虚拟服务 IP 地址，当 type=ClusterIP 时，如果不指定，则系统进行自动分配，也可以手工指定；当 type=LoadBalancer 时，则需要指定
spec.sessionAffinity	string		是否支持 Session，可选值为 ClientIP，默认为空。 ClientIP: 表示将同一个客户端（根据客户端的 IP 地址决定）的访问请求都转发到同一个后端 Pod
spec.ports[]	list		Service 需要暴露的端口列表
spec.ports[].name	string		端口名称
spec.ports[].protocol	string		端口协议，支持 TCP 和 UDP，默认为 TCP
spec.ports[].port	int		服务监听的端口号
spec.ports[].targetPort	int		需要转发到后端 Pod 的端口号
spec.ports[].nodePort	int		当 spec.type=NodePort 时，指定映射到物理机的端口号
Status	object		当 spec.type=LoadBalancer 时，设置外部负载均衡器的地址，用于公有云环境
status.loadBalancer	object		外部负载均衡器
status.loadBalancer.ingress	object		外部负载均衡器
status.loadBalancer.ingress.ip	string		外部负载均衡器的 IP 地址
status.loadBalancer.ingress.hostname	string		外部负载均衡器的主机名

2.5.2 Service基本用法

一般来说，对外提供服务的应用程序需要通过某种机制来实现，对于容器应用最简便的方式就是通过TCP/IP机制及监听IP和端口号来实现。例如，我们定义一个提供Web服务的RC，由两个tomcat容器副本组成，每个容器通过containerPort设置提供服务的端口号为8080：

```
webapp-rc.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: webapp
spec:
  replicas: 2
  template:
    metadata:
      name: webapp
      labels:
        app: webapp
    spec:
      containers:
        - name: webapp
          image: tomcat
```

```
ports:
  -containerPort: 80
```

创建该RC:

```
# kubectl create -f webapp-rc.yaml
replicationcontroller "webapp" created
```

获取Pod的IP地址:

```
# kubectl get pods -l app=webapp -o yaml | grep podIP
podIP: 172.17.1.4
podIP: 172.17.1.3
```

访问这两个Pod提供的Tomcat服务:

```
# curl 172.17.1.3:8080
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Apache Tomcat/8.0.35</title>
    .....
# curl 172.17.1.4:8080
<!DOCTYPE html>
<html lang="en">
  <head>
```

```
<meta charset="UTF-8" />
<title>Apache Tomcat/8.0.35</title>
.....
```

直接通过Pod的IP地址和端口号可以访问容器应用，但是Pod的IP地址是不可靠的，例如当Pod所在的Node发生故障时，Pod将被Kubernetes重新调度到另一台Node进行启动，Pod的IP地址将发生变化。更重要的是，如果容器应用本身是分布式的部署方式，通过多个实例共同提供服务，就需要在这些实例的前端设置一个负载均衡器来实现请求的分发。Kubernetes中的Service就是设计出来用于解决这些问题的核心组件。

以前面创建的webapp应用为例，为了让客户端应用能够访问到两个Tomcat Pod实例，需要创建一个Service来提供服务。Kubernetes提供了一种快速的方法，即通过kubect1 expose命令来创建Service:

```
# kubect1 expose rc webapp
service "webapp" exposed
```

查看新创建的Service，可以看到系统为它分配了一个虚拟的IP地址（ClusterIP），而Service所需的端口号则从Pod中的containerPort复制而来:

```
# kubect1 get svc
```

	NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)
AGE				
	webapp	169.169.235.79	<none>	8080/TCP

接下来，我们就可以通过Service的IP地址和Service的端口号访问该Service了：

```
# curl 169.169.235.79:8080

<!DOCTYPE html>

<html lang="en">

    <head>

        <meta charset="UTF-8" />

        <title>Apache Tomcat/8.0.35</title>

        .
        .
        .
```

这里，对Service地址169.169.235.79: 8080的访问被自动负载分发到了后端两个Pod之一：172.17.1.3: 8080或172.17.1.4: 8080。

除了使用`kubectl expose`命令创建Service，我们也可以通过配置文件定义Service，再通过`kubectl create`命令进行创建。例如对于前面的webapp应用，我们可以设置一个Service，代码如下：

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
spec:
  ports:
    - port: 8081
      targetPort: 8080
```

```
selector:
  app: webapp
```

Service定义中的关键字段是ports和selector。本例中ports定义部分指定了Service所需的虚拟端口号为8081，由于与Pod容器端口号8080不一样，所以需要再通过targetPort来指定后端Pod的端口号。selector定义部分设置的是后端Pod所拥有的label: app=webapp。

创建该Service并查看其ClusterIP地址:

```
# kubectl create -f webapp-svc.yaml
service "webapp" created

# kubectl get svc
```

	NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)
AGE				
	webapp	169.169.28.190	<none>	8081/TCP

通过Service的IP地址和Service的端口号进行访问:

```
# curl 169.169.28.190:8081
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Apache Tomcat/8.0.35</title>
    .....
  </head>
</html>
```

同样，对Service地址169.169.28.190: 8081的访问被自动负载分发到了后端两个Pod之一：172.17.1.3: 8080或172.17.1.4: 8080。目前Kubernetes提供了两种负载分发策略：RoundRobin和SessionAffinity，具体说明如下。

- **RoundRobin**: 轮询模式，即轮询将请求转发到后端的各个Pod上。
- **SessionAffinity**: 基于客户端IP地址进行会话保持的模式，即第1次将某个客户端发起的请求转发到后端的某个Pod上，之后从相同的客户端发起的请求都将被转发到后端相同的Pod上。

在默认情况下，Kubernetes采用RoundRobin模式进行路由选择，但我们也可以通过将service.spec.sessionAffinity设置为“ClientIP”来启用SessionAffinity策略，这样，同一个客户端发来的请求就会建立一个Session，并且对应到后端固定的某个Pod上了。

在某些应用场景中，开发人员希望自己控制负载均衡的策略，不使用Service提供的默认负载均衡的功能，Kubernetes通过Headless Service的概念来实现这种功能，即不给Service设置ClusterIP（无入口IP地址），而仅通过Label Selector将后端的Pod列表返回给调用的客户端。例如：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
```

```
spec:
  ports:
    - port: 80
  clusterIP: None
  selector:
    app: nginx
```

该Service没有虚拟的ClusterIP地址，对其进行访问将获得具有Label“app=nginx”的全部Pod列表，然后客户端程序需要实现自己的负载分发策略，再确定访问具体哪一个后端的Pod。

在某些环境中，应用系统需要将一个外部数据库作为后端服务进行连接，或将另一个集群或Namespace中的服务作为服务的后端，这时可以通过创建一个无Label Selector的Service来实现：

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

通过该定义创建的是一个不带标签选择器的Service，即无法选择后端的Pod，系统不会自动创建Endpoint，因此需要手动创建一个和该

Service同名的Endpoint，用于指向实际的后端访问地址。创建Endpoint的配置文件内容如下：

```
kind: Endpoints
apiVersion: v1
metadata:
  name: my-service
subsets:
- addresses:
  - IP: 1.2.3.4
  ports:
  - port: 80
```

如图2.16所示，访问没有标签选择器的Service和带有标签选择器的Service一样，请求将会被路由到由用户手动定义的后端Endpoint上。

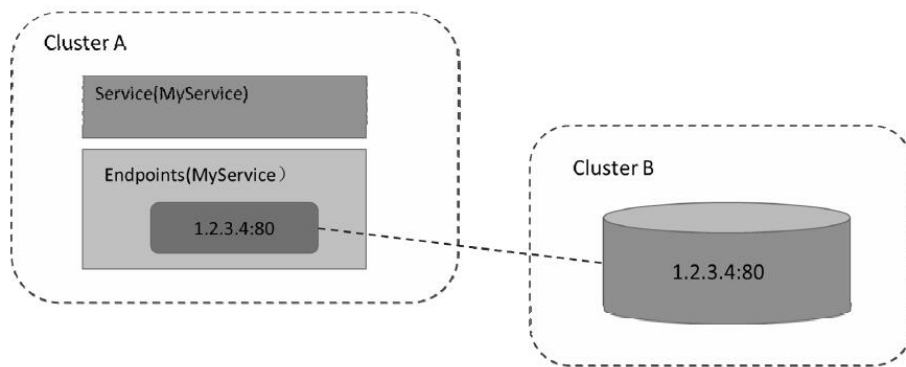


图2.16 Service指向外部服务

有时，一个容器应用也可能提供多个端口的服务，所以在Service的定义中也可以相应地设置为多个端口。在下面的例子中，Service设置了两个端口号，并且为每个端口号进行了命名：

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
spec:
  ports:
    - port: 8080
      targetPort: 8080
      name: web
    - port: 8005
      targetPort: 8005
      name: management
  selector:
    app: webapp
```

另一个例子是两个端口号使用了不同的4层协议，即TCP或UDP：

```
apiVersion: v1
kind: Service
metadata:
  name: kube-dns
  namespace: kube-system
labels:
```

```
k8s-app: kube-dns
kubernetes.io/cluster-service: "true"
kubernetes.io/name: "KubeDNS"
spec:
  selector:
    k8s-app: kube-dns
  clusterIP: 169.169.0.100
  ports:
    - name: dns
      port: 53
      protocol: UDP
    - name: dns-tcp
      port: 53
      protocol: TCP
```

2.5.3 集群外部访问Pod或服务

由于Pod和服务是Kubernetes集群范围内的虚拟概念，所以集群外的客户端系统无法通过Pod的IP地址或者服务的虚拟IP地址和虚拟端口号访问到它们。为了让外部客户端可以访问这些服务，可以将Pod或服务的端口号映射到宿主机，以使得客户端应用能够通过物理机访问容器应用。

1. 将容器应用的端口号映射到物理机

(1) 通过设置容器级别的hostPort，将容器应用的端口号映射到物理机上：

```
pod-hostport.yaml
apiVersion: v1
kind: Pod
metadata:
  name: webapp
  labels:
    app: webapp
spec:
  containers:
  - name: webapp
    image: tomcat
```

```
ports:
  - containerPort: 8080
    hostPort: 8081
```

通过kubect create命令创建这个Pod:

```
# kubectl create -f pod-hostport.yaml
pod "webapp" created
```

通过物理机的IP地址和8081端口号访问Pod内的容器服务:

```
# curl 192.168.18.3:8081
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Apache Tomcat/8.0.35</title>
    .....
  </head>
</html>
```

(2) 通过设置Pod级别的hostNetwork=true，该Pod中所有容器的端口号都将被直接映射到物理机上。设置hostNetwork=true时需要注意，在容器的ports定义部分如果不指定hostPort，则默认hostPort等于containerPort，如果指定了hostPort，则hostPort必须等于containerPort的值。

```
pod-hostnetwork.yaml
apiVersion: v1
```

```
kind: Pod
metadata:
  name: webapp
  labels:
    app: webapp
spec:
  hostNetwork: true
  containers:
  - name: webapp
    image: tomcat
    imagePullPolicy: Never
    ports:
    - containerPort: 8080
```

创建这个Pod:

```
# kubectl create -f pod-hostnetwork.yaml
pod "webapp" created
```

通过物理机的IP地址和8080端口号访问Pod内的容器服务:

```
# curl 192.168.18.4:8080
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
```


<title>Apache Tomcat/8.0.35</title>

.....

2.将Service的端口号映射到物理机

(1) 通过设置nodePort映射到物理机，同时设置Service的类型为NodePort:

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
spec:
  type: NodePort
  ports:
    - port: 8080
      targetPort: 8080
      nodePort: 8081
  selector:
    app: webapp
```

创建这个Service:

```
# kubectl create -f webapp-svc-nodeport.yaml

You have exposed your service on an external port on
all nodes in your
cluster. If you want to expose this service to the
```

```
external internet, you may
    need to set up firewall rules for the service port(s)
(tcp:8081) to serve traffic.
```

```
See http://releases.k8s.io/release-1.3/docs/user-
guide/services-firewalls.md for more details.
```

```
service "webapp" created
```

系统提示信息说明：由于要使用物理机的端口号，所以需要在防火墙上做好相应的配置，以使得外部客户端能够访问到该端口。

通过物理机的IP地址和nodePort 8081端口号访问服务：

```
# curl 192.168.18.3:8081
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Apache Tomcat/8.0.35</title>
  .....
```

同样，对该Service的访问也将被负载分发到后端的多个Pod上。

(2) 通过设置LoadBalancer映射到云服务商提供的LoadBalancer地址。这种用法仅用于在公有云服务提供商的云平台上设置Service的场景。在下面的例子中，status.loadBalancer.ingress.ip 设置的146.148.47.155为云服务商提供的负载均衡器的IP地址。对该Service的

访问请求将会通过LoadBalancer转发到后端Pod上，负载分发的实现方式则依赖于云服务商提供的LoadBalancer的实现机制。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
      nodePort: 30061
  clusterIP: 10.0.171.239
  loadBalancerIP: 78.11.24.19
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
      - ip: 146.148.47.155
```

2.5.4 DNS服务搭建指南

根据第1章对Service概念的说明，为了能够通过服务的名字在集群内部进行服务的相互访问，需要创建一个虚拟的DNS服务来完成服务名到ClusterIP的解析。本节将对如何搭建DNS服务进行详细说明。

Kubernetes提供的虚拟DNS服务名为skydns，由四个组件组成。

- (1) etcd: DNS存储。
- (2) kube2sky: 将Kubernetes Master中的Service（服务）注册到etcd。
- (3) skyDNS: 提供DNS域名解析服务。
- (4) healthz: 提供对skydns服务的健康检查功能。

图2.17描述了KubernetesDNS服务的总体架构。

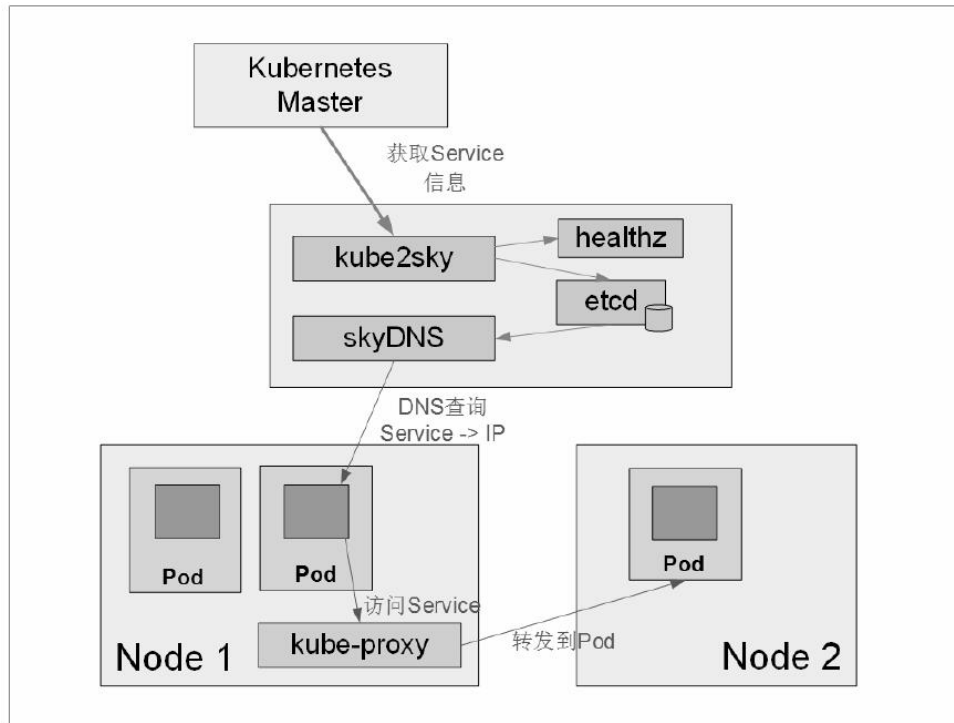


图2.17 Kubernetes DNS服务的总体架构

1.skydns配置文件说明

skydns服务由一个RC和一个Service的定义组成，分别由配置文件skydns-rc.yaml和skydns-svc.yaml定义。

skydns的RC配置文件skydns-rc.yaml的内容如下，包含了4个容器的定义：

```
skydns-rc.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: kube-dns-v11
```

```
namespace: kube-system
labels:
  k8s-app: kube-dns
  version: v11
  kubernetes.io/cluster-service: "true"
spec:
  replicas: 1
  selector:
    k8s-app: kube-dns
    version: v11
  template:
    metadata:
      labels:
        k8s-app: kube-dns
        version: v11
        kubernetes.io/cluster-service: "true"
    spec:
      containers:
        - name: etcd
          image: gcr.io/google_containers/etcd-
amd64:2.2.1
          resources:
            limits:
              cpu: 100m
              memory: 50Mi
            requests:
              cpu: 100m
```

```
        memory: 50Mi
    command:
    - /usr/local/bin/etcd
    - -data-dir
    - /tmp/data
    - -listen-client-urls
    - http://127.0.0.1:2379,http://127.0.0.1:4001
    - -advertise-client-urls
    - http://127.0.0.1:2379,http://127.0.0.1:4001
    - -initial-cluster-token
    - skydns-etcd
    volumeMounts:
    - name: etcd-storage
      mountPath: /tmp/data
    - name: kube2sky
      image: gcr.io/google_containers/kube2sky-
amd64:1.15
```

```
resources:
  limits:
    cpu: 100m
    # Kube2sky watches all pods.
    memory: 50Mi
  requests:
    cpu: 100m
    memory: 50Mi
livenessProbe:
  httpGet:
```

```
    path: /healthz
    port: 8080
    scheme: HTTP
    initialDelaySeconds: 60
    timeoutSeconds: 5
    successThreshold: 1
    failureThreshold: 5
  readinessProbe:
    httpGet:
      path: /readiness
      port: 8081
      scheme: HTTP
      # we poll on pod startup for the Kubernetes
master service and
      # only setup the /readiness HTTP server once
that's available.
    initialDelaySeconds: 30
    timeoutSeconds: 5
  args:
    # command = "/kube2sky"
    - --kube-master-url=http://192.168.18.3:8080
    - --domain=cluster.local
  - name: skydns
    image: gcr.io/google_containers/skydns:2015-
10-13-8c72f8c
  resources:
    limits:
```



```

        cpu: 100m
        memory: 50Mi
    requests:
        cpu: 100m
        memory: 50Mi
    args:
    # command = "/skydns"
    - -machines=http://127.0.0.1:4001
    - -addr=0.0.0.0:53
    - -ns-rotate=false
    - -domain=cluster.local
    ports:
    - containerPort: 53
      name: dns
      protocol: UDP
    - containerPort: 53
      name: dns-tcp
      protocol: TCP
    - name: healthz

image:
gcr.io/google_containers/exechealthz:1.0
resources:
    # keep request = limit to keep this
    container in guaranteed class
    limits:
        cpu: 10m
        memory: 20Mi

```

```

        requests:
            cpu: 10m
            memory: 20Mi
        args:
            - -cmd=nslookup
kubernetes.default.svc.cluster.local 127.0.0.1 >/dev/null
        - -port=8080
    ports:
        - containerPort: 8080
          protocol: TCP
    volumes:
        - name: etcd-storage
          emptyDir: {}
    dnsPolicy: Default # Don't use cluster DNS.

```

需要修改的几个配置参数如下。

(1) kube2sky 容器需要访问 Kubernetes Master，需要配置 Master 所在物理主机的 IP 地址和端口号，本例中设置参数 `--kube_master_url` 的值为 `http://192.168.18.3: 8080`。

(2) kube2sky 容器和 skydns 容器的启动参数 `--domain`，设置 Kubernetes 集群中 Service 所属的域名，本例中为“cluster.local”。启动后，kube2sky 会通过 API Server 监控集群中全部 Service 的定义，生成相应的记录并保存到 etcd 中。kube2sky 为每个 Service 生成以下两条记录。

- `<service_name>.<namespace_name>.<domain>`。
- `<service_name>.<namespace_name>.svc.<domain>`。。

(3) skydns的启动参数-addr=0.0.0.0: 53表示使用本机TCP和UDP的53端口提供服务。

skydns的Service配置文件skydns-svc.yaml的内容如下:

```
skydns-svc.yaml
apiVersion: v1
kind: Service
metadata:
  name: kube-dns
  namespace: kube-system
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "KubeDNS"
spec:
  selector:
    k8s-app: kube-dns
  clusterIP: 169.169.0.100
  ports:
    - name: dns
      port: 53
      protocol: UDP
    - name: dns-tcp
      port: 53
      protocol: TCP
```

注意，skydns服务使用的clusterIP需要我们指定一个固定的IP地址，每个Node的kubelet进程都将使用这个IP地址，不能通过Kubernetes自动分配。

另外，这个IP地址需要在kube-apiserver启动参数--service-cluster-ip-range指定的IP地址范围内。

在创建skydns容器之前，先修改每个Node上kubelet的启动参数。

2.修改每台Node上的kubelet启动参数

修改每台Node上kubelet的启动参数，加上以下两个参数。

- --cluster_dns=169.169.0.100: 为DNS服务的ClusterIP地址。
- --cluster_domain=cluster.local: 为DNS服务中设置的域名。

然后重启kubelet服务。

3.创建skydnsRC和Service

通过kubectl create完成skydns的RC和Service的创建:

```
#kubectl create -f skydns-rc.yaml
#kubectl create -f skydns-svc.yaml
```

查看RC、Pod和Service，确保容器成功启动:

```
# kubectl get rc --namespace=kube-system
```

NAME	DESIRED
------	---------

CURRENT	AGE	
kube-dns-v11	1	1
1d		

```
# kubectl get pods --namespace=kube-system
```

NAME	READY
STATUS	RESTARTS
AGE	
kube-dns-v11-6d1wu	4/4
Running	0
1d	

```
# kubectl get services --namespace=kube-system
```

NAME	CLUSTER-IP	EXTERNAL-IP
PORT(S)	AGE	
kube-dns	169.169.0.100	<none>
53/UDP, 53/TCP	1d	

然后，我们为redis-master应用创建一个Service:

[redis-master-service.yaml](#)

```
apiVersion: v1
kind: Service
metadata:
  name: redis-master
  labels:
    name: redis-master
spec:
```

```
ports:
- port: 6379
  targetPort: 6379
selector:
  name: redis-master
```

查看创建好的redis-master service:

```
# kubectl get services
```

NAME	CLUSTER-IP	EXTERNAL-IP
redis-master	169.169.8.10	<none>
6379/TCP	1h	

可以看到，系统为redis-master服务分配了一个虚拟IP地址：169.169.8.10。

到此，在Kubernetes集群内的虚拟DNS服务就搭建好了。在需要访问redis-master的应用中，仅需要配置上redis-master Service的名称和服务的端口号，就能够访问到redis-master应用了，让我们回顾一下redis-slave应用需要访问redis-master的配置内容：

redis-slave镜像的启动脚本/run.sh的内容为：

```
if [[ ${GET_HOSTS_FROM:-dns} == "env" ]]; then
    redis-server --slaveof ${REDIS_MASTER_SERVICE_HOST}
6379
else
```

```
redis-server --slaveof redis-master 6379  
fi
```

在使用DNS模式的情况下，redis-slave配置的Master地址为：redis-master: 6379。通过服务名进行配置，能够极大地简化客户端应用对后端服务变化的感知，包括服务虚拟IP地址的变化、服务后端Pod的变化等，对应用程序的微服务架构实现提供了强有力的支撑。

4.通过DNS查找Service

接下来使用一个带有nslookup工具的Pod来验证DNS服务是否能够正常工作：

busybox.yaml

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: busybox  
  namespace: default  
spec:  
  containers:  
    - name: busybox  
      image: gcr.io/google_containers/busybox  
      command:  
        - sleep  
        - "3600"
```

运行`kubectl create-f busybox.yaml`完成创建。

在该容器成功启动后，通过`kubectl exec<container_id>nslookup`进行测试：

```
# kubectl exec busybox -- nslookup redis-master
Server:      169.169.0.100
Address 1: 169.169.0.100

Name:        redis-master
Address 1: 169.169.8.10
```

可以看到，通过DNS服务器169.169.0.100成功找到了名为“redis-master”服务的IP地址：169.169.8.10。

如果某个Service属于不同的命名空间，那么在进行Service查找时，需要带上namespace的名字。下面以查找kube-dns服务为例：

```
# kubectl exec busybox -- nslookup kube-dns.kube-system
Server:      169.169.0.100
Address 1: 169.169.0.100

Name:        kube-dns.kube-system
Address 1: 169.169.0.100
```

如果仅使用“kube-dns”来进行查找，则将会失败：

```
nslookup: can't resolve 'kube-dns'
```

5.DNS服务的工作原理解析

让我们看看DNS服务背后的工作原理。

(1) kube2sky容器应用通过调用Kubernetes Master的API获得集群中所有Service的信息，并持续监控新Service的生成，然后写入etcd中。

查看etcd中存储的Service信息:

```
# kubectl exec kube-dns-v8-5tpm2 -c etcd --
namespace=kube-system etcdctl ls /skydns/local/cluster
/skydns/local/cluster/default
/skydns/local/cluster/svc
/skydns/local/cluster/kube-system
```

可以看到在skydns键下面，根据我们配置的域名（cluster.local）生成了local/cluster子键，接下来是namespace（default和kube-system）和svc（下面也按namespace生成子键）。

查看redis-master服务对应的键值:

```
# kubectl exec kube-dns-v8-5tpm2 -c etcd --
namespace=kube-system etcdctl get
/skydns/local/cluster/default/redis-master
```

```
{"host": "169.169.8.10", "priority": 10, "weight": 10, "ttl": 30, "targetstrip": 0}
```

可以看到，redis-master 服务对应的完整域名为 redis-master.default.cluster.local，并且其IP地址为169.169.8.10。

(2) 根据kubelet启动参数的设置（--cluster_dns），kubelet会在每个新创建的Pod中设置DNS域名解析配置文件/etc/resolv.conf文件，在其中增加了一条nameserver配置和一条search配置：

```
nameserver 169.169.0.100
search default.svc.cluster.local svc.cluster.local
cluster.local localdomain
```

通过名字服务器169.169.0.100访问的实际上就是skydns在53端口上提供的DNS解析服务。

(3) 最后，应用程序就能够像访问网站域名一样，仅仅通过服务的名字就能访问到服务了。

仍然以redis-slave为例，假设已经启动了redis-slave Pod，登录redis-slave容器进行查看，可以看到其通过DNS域名服务找到了redis-master的IP地址169.169.8.10，并成功建立了连接。

2.5.5 Ingress: HTTP 7层路由机制

根据前面对Service的使用说明，我们知道Service的表现形式为IP: Port，即工作在TCP/IP层。而对于基于HTTP的服务来说，不同的URL地址经常对应到不同的后端服务或者虚拟服务器（Virtual Host），这些应用层的转发机制仅通过Kubernetes的Service机制是无法实现的。Kubernetes v1.1版本中新增的Ingress将不同URL的访问请求转发到后端不同的Service，实现HTTP层的业务路由机制。在Kubernetes集群中，Ingress的实现需要通过Ingress的定义与Ingress Controller的定义结合起来，才能形成完整的HTTP负载分发功能。

图2.18显示了一个典型HTTP层路由的例子。

- 对 `http://mywebsite.com/api` 的访问将被路由到后端名为“api”的Service。
- 对 `http://mywebsite.com/web` 的访问将被路由到后端名为“web”的Service。
- 对 `http://mywebsite.com/doc` 的访问将被路由到后端名为“doc”的Service。

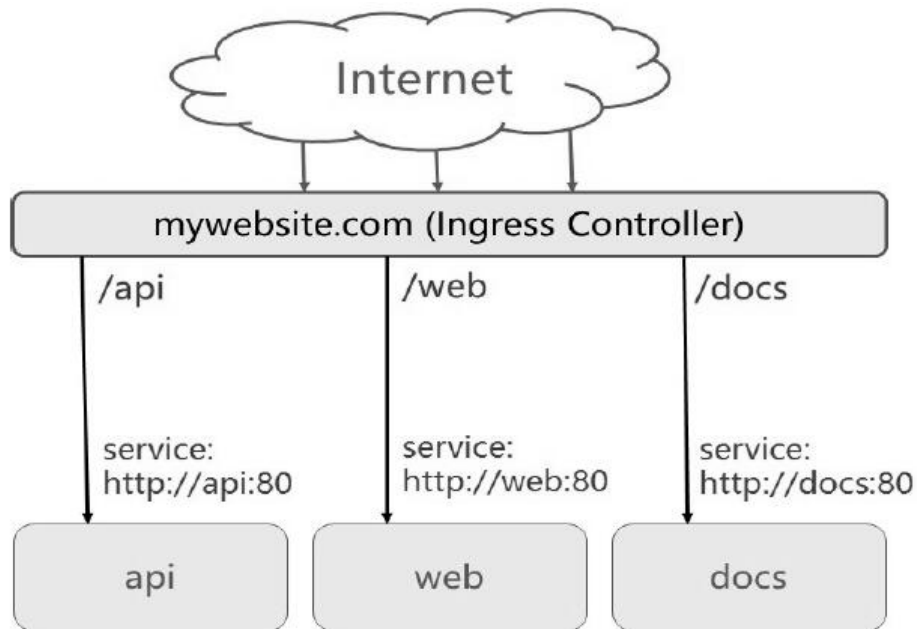


图2.18 Ingress示例

1.创建Ingress Controller

在定义Ingress之前，需要先部署Ingress Controller，以实现为所有后端Service提供一个统一的入口。Ingress Controller需要实现基于不同HTTP URL向后转发的负载分发规则，通常应该根据应用系统的需求进行个性化实现。如果公有云服务商能够提供该类型的HTTP路由LoadBalancer，则也可设置其为Ingress Controller。

在Kubernetes中，Ingress Controller将以Pod的形式运行，监控apiserver的/ingress接口（在1.3版本中为/apis/extensions/v1beta1/namespaces/<namespace_name>/ingresses接口）后端的backend services，如果service发生变化，则Ingress Controller应自动更新其转发规则。

在下面的例子中，我们使用Nginx来实现一个Ingress Controller，需要实现的基本逻辑如下。

(1) 监听apiserver，获取全部ingress的定义。

(2) 基于 ingress 的定义，生成 Nginx 所需的配置文件/etc/nginx/nginx.conf。

(3) 执行nginx-s reload命令，重新加载nginx.conf配置文件的内容，

基于Go语言的代码实现如下：

```
    for {
        rateLimiter.Accept()

        ingresses, err :=
ingClient.List(labels.Everything(), fields.Everything())
        if err != nil ||
reflect.DeepEqual(ingresses.Items, known.Items) {
            continue
        }
        if w, err := os.Create("/etc/nginx/nginx.conf");
err != nil {
            log.Fatalf("Failed to open %v: %v", nginxConf,
err)
        } else if err := tmpl.Execute(w, ingresses); err
!= nil {
            log.Fatalf("Failed to write template %v", err)
```

```
    }  
    shellOut("nginx -s reload")  
}
```

我们可以通过直接下载谷歌提供的nginx-ingress镜像来创建Ingress Controller:

```
nginx-ingress-rc.yaml  
apiVersion: v1  
kind: ReplicationController  
metadata:  
  name: nginx-ingress  
  labels:  
    app: nginx-ingress  
spec:  
  replicas: 1  
  selector:  
    app: nginx-ingress  
  template:  
    metadata:  
      labels:  
        app: nginx-ingress  
    spec:  
      containers:  
        - image: gcr.io/google_containers/nginx-  
ingress:0.1  
          name: nginx
```

```
ports:
  - containerPort: 80
    hostPort: 80
```

这里，该Nginx应用设置了hostPort，即它将容器应用监听的80端口号映射到物理机，以使得客户端应用可以通过URL地址“http://物理机IP: 80”来访问该Ingress Controller。

通过kubectl create命令创建该RC:

```
# kubectl create -f nginx-ingress-rc.yaml
replicationcontroller "nginx-ingress" created

# kubectl get pods
```

	NAME	READY	STATUS	RESTARTS
AGE				
	nginx-ingress-mrwztz	1/1	Running	0
2s				

2.定义Ingress

为mywebsite.com定义Ingress，设置到后端Service的转发规则:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: mywebsite-ingress
```

```
spec:
  rules:
    - host: mywebsite.com
      http:
        paths:
          - path: /web
            backend:
              serviceName: webapp
              servicePort: 80
```

这个Ingress的定义说明对目标URL `http://mywebsite.com/web`的访问将被转发到Kubernetes的一个Service上: `webapp: 80`。

创建该Ingress:

```
# kubectl create -f ingress.yaml
ingress "mywebsite-ingress" created

# kubectl get ingress
```

	NAME	HOSTS	ADDRESS	PORTS
AGE	mywebsite-ingress	mywebsite.com		80
17s				

在该Ingress成功创建后, 登录`nginx-ingress Pod`, 查看其自动生成的`nginx.conf`配置文件内容:

```
events {
    worker_connections 1024;
}
http {
    server {
        listen 80;
        server_name mywebsite.com;    # Ingress中定义的虚拟
host名
        resolver 127.0.0.1;

        location /web {                # Ingress中定义的路
径 /web
            proxy_pass http://webapp;  # service名
        }
    }
}
```

3.访问<http://mywebsite.com/web>

由于Ingress Controller设置了hostPort，所以我们可以通过其所在的物理机对其进行访问。可以在物理机上设置mywebsite.com对应的IP地址，也可以通过curl--resolve进行指定：

```
$ curl --resolve mywebsite.com:80:192.168.18.3
mywebsite.com/foo
```

将获得Kubernetes Service“webapp: 80”提供的主页。

4.Ingress的发展路线

当前的Ingress还是beta版本，在Kubernetes的后续版本中将增加至少以下功能。

- 支持更多TLS选项，例如SNI、重加密等。
- 支持L4和L7负载均衡策略（目前只支持HTTP层的规则）。
- 支持更多的转发规则（目前仅支持基于URL路径的），例如重定向规则、会话保持规则等。

第3章 Kubernetes核心原理

本章对Kubernetes的核心原理进行深入分析，首先从API Server的访问开始讲起，然后分析Master节点上Controller Manager各个组件的功能实现，以及Scheduler预选算法和优选算法。接下来讲解Node节点上的kubelet和kube-proxy组件的运行机制。最后，深入分析安全机制和网络原理。

3.1 Kubernetes API Server原理分析

总体来看，Kubernetes API Server的核心功能是提供了Kubernetes各类资源对象（如Pod、RC、Service等）的增、删、改、查及Watch等HTTPRest接口，成为集群内各个功能模块之间数据交互和通信的中心枢纽，是整个系统的数据总线和数据中心。除此之外，它还有以下一些功能特性。

- （1）是集群管理的API入口。
- （2）是资源配额控制的入口。
- （3）提供了完备的集群安全机制。

3.1.1 Kubernetes API Server概述

Kubernetes API Server通过一个名为kube-apiserver的进程提供服务，该进程运行在Master节点上。在默认情况下，kube-apiserver进程在本机的8080端口（对应参数--insecure-port）提供REST服务。我们可以同时启动HTTPS安全端口（--secure-port=6443）来启动安全机制，加强REST API访问的安全性。

通常我们可以通过命令行工具kubectl来与Kubernetes API Server交互，它们之间的接口是REST调用。为了测试和学习Kubernetes API Server所提供的接口，我们也可以使用curl命令行工具进行快速验证。

比如，我们登录Master节点，运行下面的curl命令，得到以JSON方式返回的Kubernetes API的版本信息：

```
# curl localhost:8080/api
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "192.168.18.131:6443"
```

```
    }  
  ]  
}
```

可以运行下面的命令，来查看Kubernetes API Server目前所支持的资源对象的种类：

```
# curl localhost:8080/api/v1
```

根据以上命令的输出，我们可以运行下面的curl命令，分别返回集群中的Pod列表、Service列表、RC列表等：

```
# curl localhost:8080/api/v1/pods  
# curl localhost:8080/api/v1/services  
# curl localhost:8080/api/v1/replicationcontrollers
```

如果我们只想对外暴露部分REST服务，则可以在Master或其他任何节点上通过运行kubect proxy进程启动一个内部代理来实现。

运行下面的命令，我们在8001端口启动代理，并且拒绝客户端访问RC的API：

```
#      kubectl      proxy      --reject-  
paths="/api/v1/replicationcontrollers"  --port=8001 --v=2  
Starting to serve on 127.0.0.1:8001
```

运行下面的命令进行验证：

```
# curl localhost:8001/api/v1/replicationcontrollers
```

<h3>Unauthorized</h3>

kubect proxy具有很多特性，最实用的一个特性是提供简单有效的安全机制，比如采用白名单来限制非法客户端访问时，只要增加下面这个参数即可：

```
--accept-hosts="^localhost$,^127\.\.0\.\.0\.\.1$,^\\[::1\\]$"
```

最后一种方式是通过编程的方式调用Kubernetes API Server。具体使用场景又细分为以下两种。

第1种使用场景：运行在Pod里的用户进程调用Kubernetes API，通常用来实现分布式集群搭建的目标。比如下面这段来自谷歌官方的Elastic Search集群例子中的代码，Pod在启动的过程中通过访问Endpoints的API，找到属于elasticsearch-logging这个Service的所有Pod副本的IP地址，用来构建集群，如图3.1所示。

```
if elasticsearch == nil {
    glog.Warningf("Failed to find the elasticsearch-logging service: %v", err)
    return
}

var endpoints *api.Endpoints
addrs := []string{}
// Wait for some endpoints.
count := 0
for t := time.Now(); time.Since(t) < 5*time.Minute; time.Sleep(10 * time.Second) {
    endpoints, err = c.Endpoints(api.NamespaceSystem).Get("elasticsearch-logging")
    if err != nil {
        continue
    }
    addrs = flattenSubsets(endpoints.Subsets)
    glog.Infof("Found %s", addrs)
    if len(addrs) > 0 && len(addrs) == count {
        break
    }
    count = len(addrs)
}

glog.Infof("Endpoints = %s", addrs)
fmt.Printf("discovery.zen.ping.unicast.hosts: [%s]\n", strings.Join(addrs, ", "))

export NODE_MASTER=${NODE_MASTER:-true}
export NODE_DATA=${NODE_DATA:-true}
/elasticsearch-logging_discovery >> /elasticsearch-1.5.2/config/elasticsearch.yml
export HTTP_PORT=${HTTP_PORT:-9200}
export TRANSPORT_PORT=${TRANSPORT_PORT:-9300}
/elasticsearch-1.5.2/bin/elasticsearch
```

1 等待5分钟获取集群里其他节点的地址信息并输出到控制台，随后被写入elasticsearch的配置文件里

2 来自镜像中的容器启动脚本

图3.1 应用程序编程访问API Server

在上述使用场景中，Pod中的进程如何知道API Server的访问地址呢？答案很简单，因为Kubernetes API Server本身也是一个Service，它的名字就是“kubernetes”，并且它的Cluster IP地址是Cluster IP地址池里的第1个地址！另外，它所服务的端口是HTTPS端口443，通过kubectl get service命令可以确认这一点：

```
# kubectl get service
```

	NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)
AGE				
	kubernetes	169.169.0.1	<none>	443/TCP
30d				

第2种使用场景：开发基于Kubernetes的管理平台。比如调用Kubernetes API来完成Pod、Service、RC等资源对象的图形化创建和管理界面，此时可以使用Kubernetes及各开源社区为开发人员提供的各种语言版本的Client Library。我们会在后面介绍通过编程方式访问API Server的一些细节技术。

3.1.2 独特的Kubernetes Proxy API接口

前面我们说过，Kubernetes API Server最主要的REST接口是资源对象的增、删、改、查，除此之外，它还提供了一类很特殊的REST接口——Kubernetes Proxy API接口，这类接口的作用是代理REST请求，即Kubernetes API Server把收到的REST请求转发到某个Node上的kubelet守护进程的REST端口上，由该Kubelet进程负责响应。

首先，我们来说Kubernetes Proxy API里关于Node的相关接口，该接口的REST路径为/api/v1/proxy/nodes/{name}，其中{name}为节点的名称或IP地址，包括以下几个具体接口：

- /api/v1/proxy/nodes/{name}/pods/ #列出指定节点内所有Pod的信息
- /api/v1/proxy/nodes/{name}/stats/ #列出指定节点内物理资源的统计信息
- /api/v1/proxy/nodes/{name}/spec/ #列出指定节点的概要信息

例如当前Node节点的名字为k8s-node-1，用下面的命令即可获取该节点上所有运行中的Pod：

```
# curl localhost:8080/api/v1/proxy/nodes/k8s-node-1/pods
```

需要说明的是：这里获取的Pod的信息数据来自Node而非etcd数据库，所以两者可能在某些时间点会有偏差。此外，如果kubelet进程在

启动时包含 `--enable-debugging-handlers=true` 参数，那么 Kubernetes Proxy API 还会增加下面的接口：

- `/api/v1/proxy/nodes/{name}/run` #在节点上运行某个容器
- `/api/v1/proxy/nodes/{name}/exec` #在节点上的某个容器中运行某条命令
- `/api/v1/proxy/nodes/{name}/attach` #在节点上 attach 某个容器
`/api/v1/proxy/nodes/{name}/portForward` #实现节点上的Pod端口转发
- `/api/v1/proxy/nodes/{name}/logs` #列出节点的各类日志信息，例如 `tallylog`、`lastlog`、`wtmp`、`ppp/`、`rhsm/`、`audit/`、`tuned/` 和 `anaconda/` 等
- `/api/v1/proxy/nodes/{name}/metrics` #列出和该节点相关的Metrics信息
- `/api/v1/proxy/nodes/{name}/runningpods` #列出节点内运行中的Pod信息
- `/api/v1/proxy/nodes/{name}/debug/pprof` #列出节点内当前Web服务的状态，包括CPU占用情况和内存使用情况等

接下来，我们来说说Kubernetes Proxy API里关于Pod的相关接口，通过这些接口，我们可以访问Pod里某个容器提供的服务（如Tomcat在8080端口服务）：

- `/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path : *}` #访问Pod的某个服务接口
- `/api/v1/proxy/namespaces/{namespace}/pods/{name}` #访问Pod
- `/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path : *}` #访问Pod的某个服务接口

- `/api/v1/namespaces/{namespace}/pods/{name}/proxy` #访问Pod

在上面的4个接口里，后面两个接口的功能与前面两个完全一样，只是写法不同。下面我们用第1章的Java Web例子中的Tomcat Pod来说明上述Proxy接口的用法。

首先，得到Pod的名字：

# kubectl get pods				
	NAME	READY	STATUS	RESTARTS
AGE				
	mysql-c95jc	1/1	Running	0
8d				
	myweb-g9pmm	1/1	Running	0
8d				

然后，运行下面的命令，会输出Tomcat的首页，即相当于访问 `http://localhost: 8080/`：

	#	curl
<code>http://localhost:8080/api/v1/proxy/namespaces/default/pods/myweb-g9pmm/</code>		

我们也可以在浏览器中访问上面的地址，比如Master节点的IP地址是192.168.18.131，我们在浏览器中输入 `http://192.168.18.131:8080/api/v1/proxy/namespaces/default/pods/myweb-g9pmm/`，就能够访问Tomcat首页了；而如果输

入 `/api/v1/proxy/namespaces/default/pods/myweb-g9pmm/demo`，就能访问Tomcat中Demo应用的页面了。

看到这里，你可能明白Pod的Proxy接口的作用和意义了：在Kubernetes集群之外访问某个Pod容器的服务（HTTP服务）时，可以用Proxy API实现，这种场景多用于管理目的，比如逐一排查Service的Pod副本，检查哪些Pod的服务存在异常问题。

最后我们说说Service，Kubernetes Proxy API也有Service的Proxy接口，其接口定义与Pod的接口定义基本一样：`/api/v1/proxy/namespaces/{namespace}/services/{name}`。比如，我们想访问myweb这个Service，则可以在浏览器里输入
`http://192.168.18.131:8080/api/v1/proxy/namespaces/default/services/myweb/demo/`。

3.1.3 集群功能模块之间的通信

从图3.2中可以看出，KubernetesAPI Server作为集群的核心，负责集群各功能模块之间的通信。集群内的各个功能模块通过API Server将信息存入etcd，当需要获取和操作这些数据时，则通过API Server提供的REST接口（用GET、LIST或WATCH方法）来实现，从而实现各模块之间的信息交互。

常见的一个交互场景是kubelet进程与API Server的交互。每个Node节点上的kubelet每隔一个时间周期，就会调用一次API Server的REST接口报告自身状态，API Server接收到这些信息后，将节点状态信息更新到etcd中。此外，kubelet也通过API Server的Watch接口监听Pod信息，如果监听到新的Pod副本被调度绑定到本节点，则执行Pod对应的容器的创建和启动逻辑；如果监听到Pod对象被删除，则删除本节点上的相应的Pod容器；如果监听到修改Pod信息，则kubelet监听到变化后，会相应地修改本节点的Pod容器。

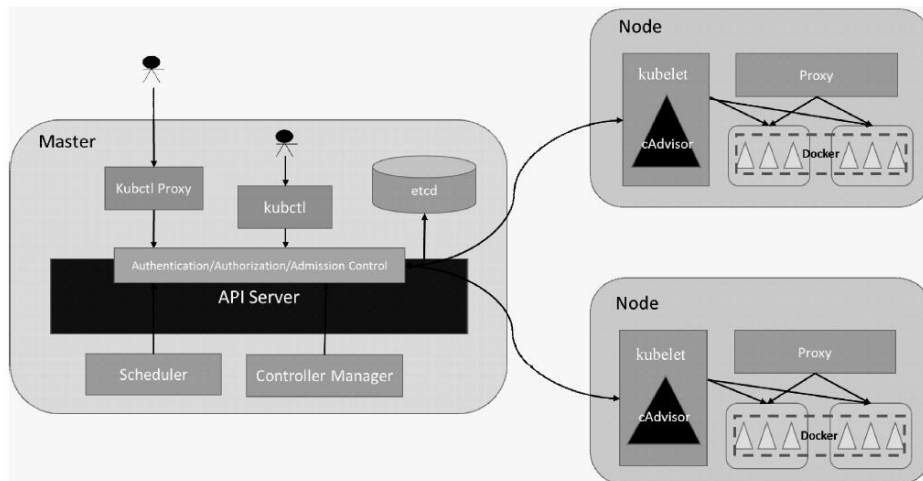


图3.2 Kubernetes结构图

另外一个交互场景是kube-controller-manager进程与API Server的交互。kube-controller-manager中的Node Controller模块通过API Server提供的Watch接口，实时监控Node的信息，并做相应处理。

还有一个比较重要的交互场景是kube-scheduler与API Server的交互。当Scheduler通过API Server的Watch接口监听到新建Pod副本的信息后，它会检索所有符合该Pod要求的Node列表，开始执行Pod调度逻辑，调度成功后将Pod绑定到目标节点上。

为了缓解集群各模块对API Server的访问压力，各功能模块都采用缓存机制来缓存数据。各功能模块定时从API Server获取指定的资源对象信息（通过LIST及WATCH方法），然后将这些信息保存到本地缓存，功能模块在某些情况下不直接访问API Server，而是通过访问缓存数据来间接访问API Server。

3.2 Controller Manager原理分析

Controller Manager作为集群内部的管理控制中心，负责集群内的Node、Pod副本、服务端点（Endpoint）、命名空间（Namespace）、服务账号（ServiceAccount）、资源定额（ResourceQuota）等的管理，当某个Node意外宕机时，Controller Manager会及时发现此故障并执行自动化修复流程，确保集群始终处于预期的工作状态。

如图3.3所示，Controller Manager内部包含Replication Controller、Node Controller、ResourceQuota Controller、Namespace Controller、ServiceAccount Controller、Token Controller、Service Controller及Endpoint Controller等多个Controller，每种Controller都负责一种具体的控制流程，而Controller Manager正是这些Controller的核心管理者。

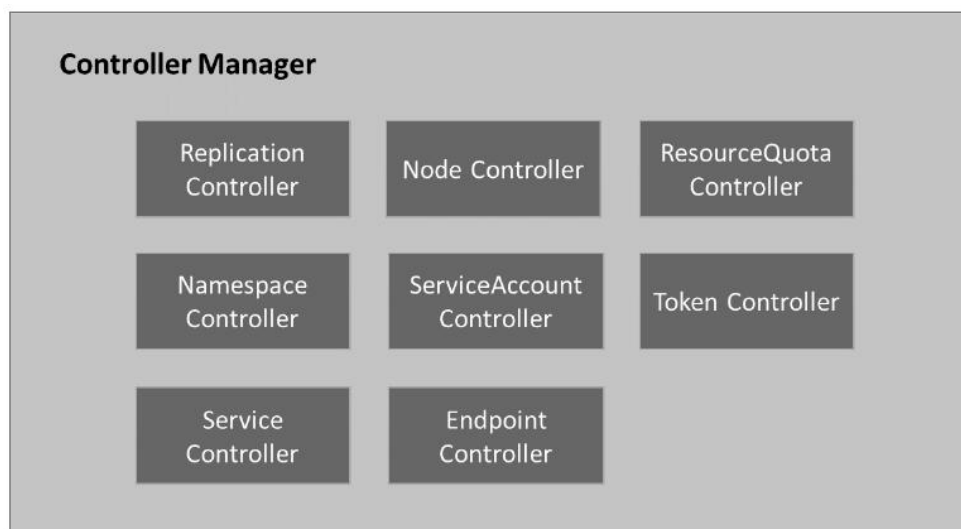


图3.3 Controller Manager结构图

一般来说，智能系统和自动系统通常会通过一个被称为“操纵系统”的机构来不断修正系统的工作状态。在Kubernetes集群中，每个Controller都是这样一个“操纵系统”，它们通过API Server提供的接口实时监控整个集群里的每个资源对象的当前状态，当发生各种故障导致系统状态发生变化时，会尝试着将系统状态从“现有状态”修正到“期望状态”。本节将详细分析Controller Manager的这些Controller的原理。

由于ServiceAccount Controller与Token Controller是与安全相关的两个控制器，并且与Service Account、Token密切相关，所以我们将对它们的分析放到后面的深入集群安全的章节中讲解。

在Kubernetes集群中与Controller Manager并重的另一个组件是Kubernetes Scheduler，它的作用是将待调度的Pod（包括通过API Server新创建的Pod及RC为补足副本而创建的Pod等）通过一些复杂的调度流程计算出最佳目标节点，然后绑定到该节点上。本章最后会介绍Kubernetes Scheduler调度器的基本原理。

3.2.1 ReplicationController

为了区分Controller Manager中的Replication Controller（副本控制器）和资源对象Replication Controller，我们将资源对象Replication Controller简写为RC，而本节中的Replication Controller是指“副本控制器”，以便于后续分析。

Replication Controller的核心作用是确保在任何时候集群中一个RC所关联的Pod副本数量保持预设值。如果发现Pod副本数量超过预期值，则Replication Controller会销毁一些Pod副本；反之，Replication Controller会自动创建新的Pod副本，直到符合条件的Pod副本数量达到预设值。需要注意的一点是：只有当Pod的重启策略是Always的时候（RestartPolicy=Always），Replication Controller才会管理该Pod的操作（例如创建、销毁、重启等）。在通常情况下，Pod对象被成功创建后不会消失，唯一的例外是当Pod处于succeeded或failed状态的时间过长（超时参数由系统设定）时，该Pod会被系统自动回收，管理该Pod的副本控制器将在其他工作节点上重新创建、运行该Pod副本。

RC中的Pod模板就像一个模具，模具制作出来的东西一旦离开模具，它们之间就再也没关系了。同样，一旦Pod被创建完毕，无论模板如何变化，甚至换成一个新的模板，也不会影响到已经创建的Pod。此外，Pod可以通过修改它的标签来实现脱离RC的管控。该方法可以用于将Pod从集群中迁移、数据修复等调试。对于被迁移的Pod副本，RC会自动创建一个新的副本替换被迁移的副本。需要注意的是，删除一个RC不会影响它所创建的Pod。如果想删除一个RC所控制的

Pod，则需要将该RC的副本数（Replicas）属性设置为0，这样所有的Pod副本都会被自动删除。

我们最好不要越过RC而直接创建Pod，因为Replication Controller会通过RC管理Pod副本，实现自动创建、补足、替换、删除Pod副本，这样能提高系统的容灾能力，减少由于节点崩溃等意外状况造成的损失。即使你的应用程序只用到一个Pod副本，我们也强烈建议使用RC来定义Pod。

我们总结一下Replication Controller的职责，如下所述。

（1）确保当前集群中有且仅有N个Pod实例，N是RC中定义的Pod副本数量。

（2）通过调整RC的spec.replicas属性值来实现系统扩容或者缩容。

（3）通过改变RC中的Pod模板（主要是镜像版本）来实现系统的滚动升级。

最后，我们总结一下Replication Controller的典型使用场景，如下所述。

（1）重新调度（Rescheduling）。如前面所提及的，不管你想运行1个副本还是1000个副本，副本控制器都能确保指定数量的副本存在于集群中，即使发生节点故障或Pod副本被终止运行等意外状况。

（2）弹性伸缩（Scaling）。手动或者通过自动扩容代理修改副本控制器的spec.replicas属性值，非常容易实现扩大或缩小副本的数量。

(3) 滚动更新 (Rolling Updates)。副本控制器被设计成通过逐个替换Pod的方式来辅助服务的滚动更新。推荐的方式是创建一个新的只有一个副本的RC，若新的RC副本数量加1，则旧的RC的副本数量减1，直到这个旧的RC的副本数量为零，然后删除该旧的RC。通过上述模式，即使在滚动更新的过程中发生了不可预料的错误，Pod集合的更新也都在可控范围内。在理想情况下，滚动更新控制器需要将准备就绪的应用考虑在内，并保证在集群中任何时刻都有足够数量的可用Pod。

3.2.2 NodeController

kubelet进程在启动时通过API Server注册自身的节点信息，并定时向API Server汇报状态信息，API Server接收到这些信息后，将这些信息更新到etcd中，etcd中存储的节点信息包括节点健康状况、节点资源、节点名称、节点地址信息、操作系统版本、Docker版本、kubelet版本等。节点健康状况包含“就绪”（True）“未就绪”（False）和“未知”（Unknown）三种。

Node Controller通过API Server实时获取Node的相关信息，实现管理和监控集群中的各个Node节点的相关控制功能，NodeController的核心工作流程如图3.4所示。

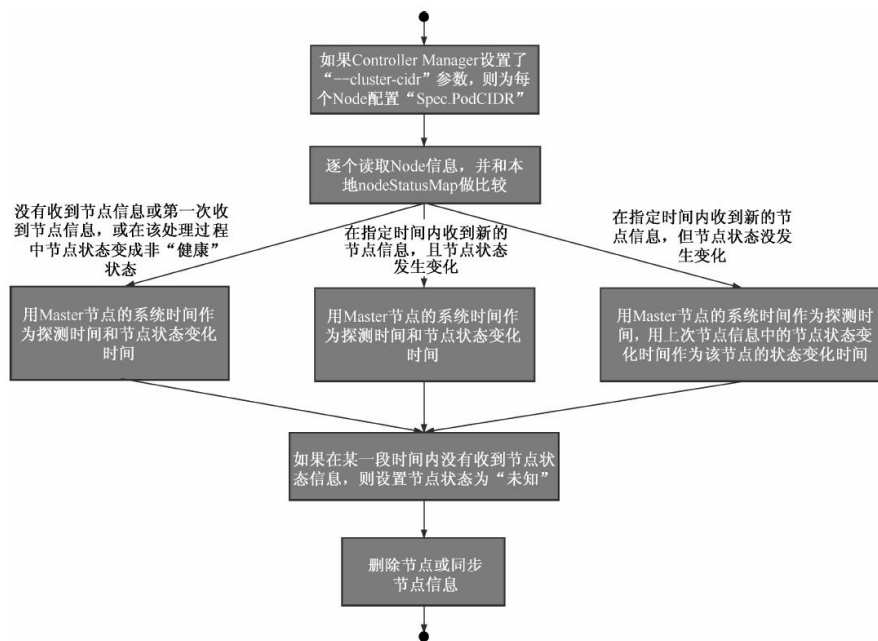


图3.4 Node Controller流程图

对流程中关键点的解释如下。

(1) **Controller Manager**在启动时如果设置了`--cluster-cidr`参数，那么为每个没有设置`Spec.PodCIDR`的**Node**节点生成一个**CIDR**地址，并用该**CIDR**地址设置节点的`Spec.PodCIDR`属性，这样做的目的是防止不同节点的**CIDR**地址发生冲突。

(2) 逐个读取节点信息，多次尝试修改**nodeStatusMap**中的节点状态信息，将该节点信息和**Node Controller**的**nodeStatusMap**中保存的节点信息做比较。如果判断出没有收到**kubelet**发送的节点信息、第1次收到节点**kubelet**发送的节点信息，或在该处理过程中节点状态变成非“健康”状态，则在**nodeStatusMap**中保存该节点的状态信息，并用**Node Controller**所在节点的系统时间作为探测时间和节点状态变化时间。如果判断出在指定时间内收到新的节点信息，且节点状态发生变化，则在**nodeStatusMap**中保存该节点的状态信息，并用**Node Controller**所在节点的系统时间作为探测时间和节点状态变化时间。如果判断出在指定时间内收到新的节点信息，但节点状态没发生变化，则在**nodeStatusMap**中保存该节点的状态信息，并用**Node Controller**所在节点的系统时间作为探测时间，用上次节点信息中的节点状态变化时间作为该节点的状态变化时间。如果判断出在某一段时间（`gracePeriod`）内没有收到节点状态信息，则设置节点状态为“未知”（**Unknown**），并且通过**API Server**保存节点状态。

(3) 逐个读取节点信息，如果节点状态变为非“就绪”状态，则将节点加入待删除队列，否则将节点从该队列中删除。如果节点状态为非“就绪”状态，且系统指定了**Cloud Provider**，则**Node Controller**调用**Cloud Provider**查看节点，若发现节点故障，则删除**etcd**中的节点信息，并删除和该节点相关的**Pod**等资源的信息。

3.2.3 ResourceQuotaController

作为完备的企业级的容器集群管理平台，Kubernetes也提供了资源配额管理（ResourceQuota Controller）这一高级功能，资源配额管理确保了指定的资源对象在任何时候都不会超量占用系统物理资源，避免了由于某些业务进程的设计或实现的缺陷导致整个系统运行紊乱甚至意外宕机，对整个集群的平稳运行和稳定性有非常重要的作用。

目前Kubernetes支持如下三个层次的资源配额管理。

- （1）容器级别，可以对CPU和Memory进行限制。
- （2）Pod级别，可以对一个Pod内所有容器的可用资源进行限制。
- （3）Namespace级别，为Namespace（多租户）级别的资源限制，包括：
 - Pod数量；
 - ReplicationController数量；
 - Service数量；
 - ResourceQuota数量；
 - Secret数量；
 - 可持有的PV（Persistent Volume）数量。

Kubernetes的配额管理是通过Admission Control（准入控制）来控制的，Admission Control当前提供了两种方式的配额约束，分别是LimitRanger与ResourceQuota。其中LimitRanger作用于Pod和Container上，而ResourceQuota则作用于Namespace上，限定一个Namespace里的各类资源的使用总额。

如图3.5所示，如果在Pod定义中同时声明了LimitRanger，则用户通过API Server请求创建或修改资源时，Admission Control会计算当前配额的使用情况，如果不符合配额约束，则创建对象失败。对于定义了ResourceQuota的Namespace，ResourceQuota Controller组件则负责定期统计和生成该Namespace下的各类对象的资源使用总量，统计结果包括Pod、Service、RC、Secret和Persistent Volume等对象实例个数，以及该Namespace下所有Container实例所使用的资源量（目前包括CPU和内存），然后将这些统计结果写入etcd的resourceQuotaStatusStorage目录（resourceQuotas/status）中。写入resourceQuotaStatusStorage的内容包含Resource名称、配额值（ResourceQuota对象中spec.hard域下包含的资源的值）、当前使用值（ResourceQuota Controller统计出来的值）。随后这些统计信息被Admission Control使用，以确保相关Namespace下的资源配额总量不会超过ResourceQuota中的限定值。

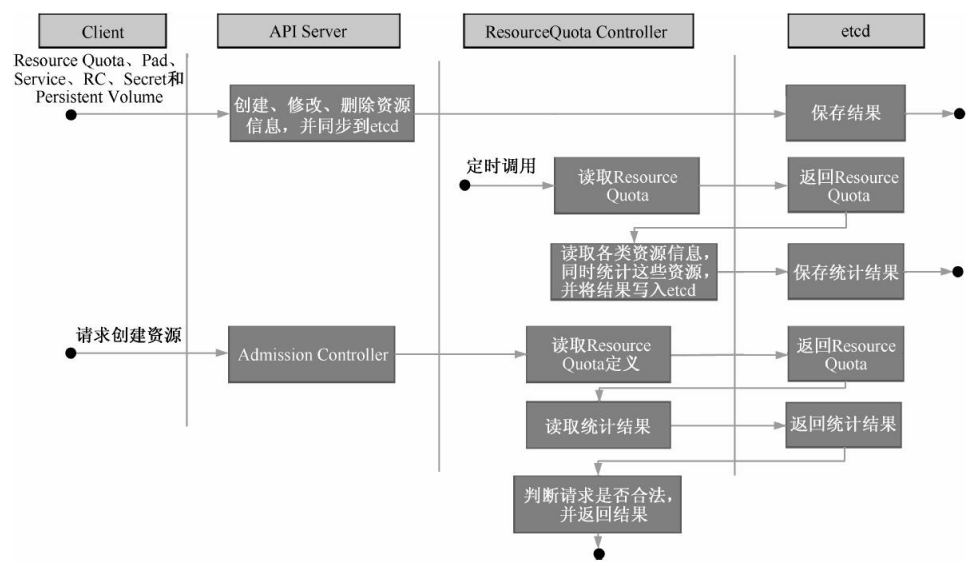


图3.5 ResourceQuota Controller流程图

3.2.4 NamespaceController

用户通过API Server可以创建新的Namespace并保存在etcd中，Namespace Controller定时通过API Server读取这些Namespace信息。如果Namespace被API标识为优雅删除（通过设置删除期限，即DeletionTimestamp属性被设置），则将该Namespace的状态设置成“Terminating”并保存到etcd中。同时Namespace Controller删除该Namespace下的ServiceAccount、RC、Pod、Secret、PersistentVolume、ListRange、ResourceQuota和Event等资源对象。

当Namespace的状态被设置成“Terminating”后，由Admission Controller的NamespaceLifecycle插件来阻止为该Namespace创建新的资源。同时，在Namespace Controller删除完该Namespace中的所有资源对象后，Namespace Controller对该Namespace执行finalize操作，删除Namespace的spec.finalizers域中的信息。

如果Namespace Controller观察到Namespace设置了删除期限，同时Namespace的spec.finalizers域值是空的，那么Namespace Controller将通过API Server删除该Namespace资源。

3.2.5 Service Controller与Endpoint Controller

我们先说说Endpoints Controller，在这之前，让我们先看看Service、Endpoints与Pod的关系，如图3.6所示，Endpoints表示了一个Service对应的所有Pod副本的访问地址，而Endpoints Controller就是负责生成和维护所有Endpoints对象的控制器。

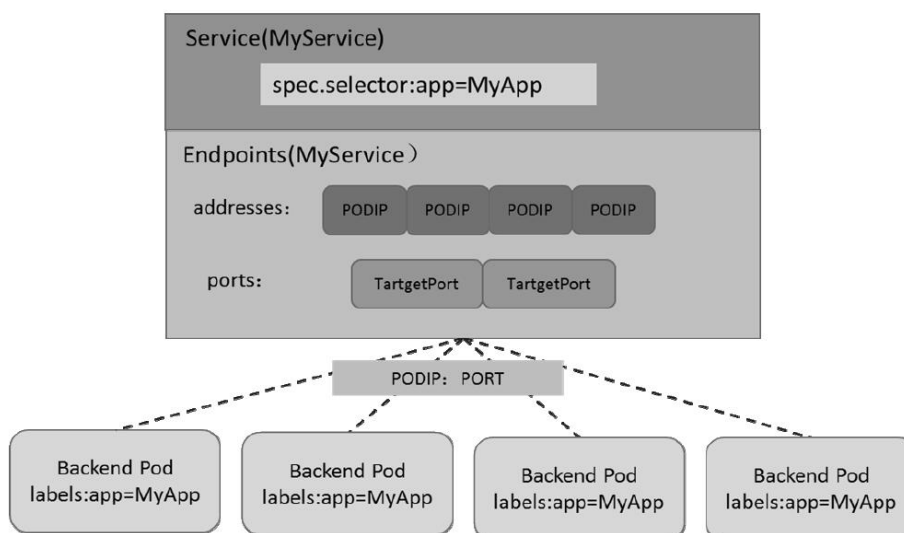


图3.6 Service、Endpoint、Pod的关系

它负责监听Service和对应的Pod副本的变化，如果监测到Service被删除，则删除和该Service同名的Endpoints对象；如果监测到新的Service被创建或者修改，则根据该Service信息获得相关的Pod列表，然后创建或者更新Service对应的Endpoints对象。如果监测到Pod的事件，则更新它所对应的Service的Endpoints对象（增加、删除或者修改对应的Endpoint条目）。

那么，Endpoints对象是在哪里被使用的呢？答案是每个Node上的kube-proxy进程，kube-proxy进程获取每个Service的Endpoints，实现了Service的负载均衡功能。在后面的章节中我们会深入讲解这部分内容。

接下来我们说说Service Controller的作用，它其实是属于Kubernetes集群与外部的云平台之间的一个接口控制器。Service Controller监听Service的变化，如果是一个LoadBalancer类型的Service（externalLoadBalancers=true），则Service Controller确保外部的云平台上该Service对应的LoadBalancer实例被相应地创建、删除及更新路由转发表（根据Endpoints的条目）。

3.3 Scheduler原理分析

我们在前面深入分析了Controller Manager及它所包含的各个组件的运行机制。本节我们将继续对Kubernetes中负责Pod调度的重要功能模块——Kubernetes Scheduler的工作原理和运行机制做深入分析。

Kubernetes Scheduler在整个系统中承担了“承上启下”的重要功能，“承上”是指它负责接收Controller Manager创建的新Pod，为其安排一个落脚的“家”——目标Node；“启下”是指安置工作完成后，目标Node上的kubelet服务进程接管后继工作，负责Pod生命周期中的“下半生”。

具体来说，Kubernetes Scheduler的作用是将待调度的Pod（API新创建的Pod、Controller Manager为补足副本而创建的Pod等）按照特定的调度算法和调度策略绑定（Binding）到集群中的某个合适的Node上，并将绑定信息写入etcd中。在整个调度过程中涉及三个对象，分别是：待调度Pod列表、可用Node列表，以及调度算法和策略。简单地说，就是通过调度算法调度为待调度Pod列表的每个Pod从Node列表中选择一个最适合的Node。

随后，目标节点上的kubelet通过API Server监听到Kubernetes Scheduler产生的Pod绑定事件，然后获取对应的Pod清单，下载Image镜像，并启动容器。完整的流程如图3.7所示。

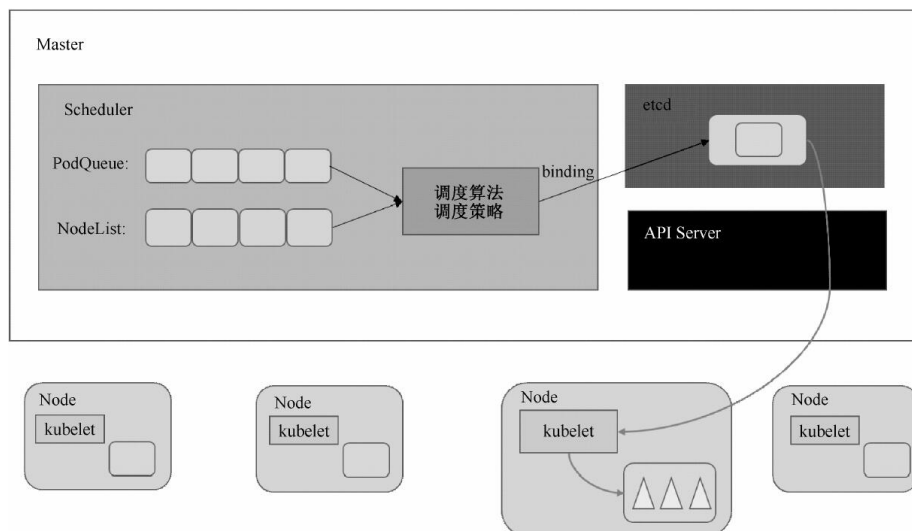


图3.7 Scheduler流程

Kubernetes Scheduler当前提供的默认调度流程分为以下两步。

(1) 预选调度过程，即遍历所有目标Node，筛选出符合要求的候选节点。为此，Kubernetes内置了多种预选策略（xxx Predicates）供用户选择。

(2) 确定最优节点，在第1步的基础上，采用优选策略（xxx Priority）计算出每个候选节点的积分，积分最高者胜出。

Kubernetes Scheduler的调度流程是通过插件方式加载的“调度算法提供者”（AlgorithmProvider）具体实现的。一个AlgorithmProvider其实就是包括了一组预选策略与一组优先选择策略的结构体，注册AlgorithmProvider的函数如下：

```
func RegisterAlgorithmProvider(name string,
predicateKeys, priorityKeys util.StringSet)
```

它包含三个参数：“name string”参数为算法名；“predicateKeys”参数为算法用到的预选策略集合；“priorityKeys”为算法用到的优选策略集合。

Scheduler 中 可 用 的 预 选 策 略 包 含：NoDiskConflict、PodFitsResources、PodSelectorMatches、PodFitsHost、CheckNodeLabelPresence、CheckServiceAffinity和PodFitsPorts策略等。其默认的AlgorithmProvider加载的预选策略Predicates包括：PodFitsPorts（PodFitsPorts）、PodFitsResources（PodFitsResources）、NoDiskConflict（NoDiskConflict）、MatchNodeSelector（PodSelectorMatches）和HostName（PodFitsHost），即每个节点只有通过前面提及的5个默认预选策略后，才能初步被选中，进入下一个流程。

下面列出的是对所有预选策略的详细说明。

1) NoDiskConflict

判断备选Pod的GCEPersistentDisk或AWSElasticBlockStore和备选的节点中已存在的Pod是否存在冲突。检测过程如下。

（1）首先，读取备选Pod的所有Volume的信息（即pod.Spec.Volumes），对每个Volume执行以下步骤进行冲突检测。

（2）如果该Volume是GCEPersistentDisk，则将Volume和备选节点上的所有Pod的每个Volume进行比较，如果发现相同的GCEPersistentDisk，则返回false，表明存在磁盘冲突，检查结束，反馈给调度器该备选节点不适合作为备选Pod；如果该Volume是AWSElasticBlockStore，则将Volume和备选节点上的所有Pod的每个

Volume 进行比较，如果发现相同的 `AWSElasticBlockStore`，则返回 `false`，表明存在磁盘冲突，检查结束，反馈给调度器该备选节点不适合备选Pod。

(3) 如果检查完备选Pod的所有Volume均未发现冲突，则返回 `true`，表明不存在磁盘冲突，反馈给调度器该备选节点适合备选Pod。

2) PodFitsResources

判断备选节点的资源是否满足备选Pod的需求，检测过程如下。

(1) 计算备选Pod和节点中已存在Pod的所有容器的需求资源（内存和CPU）的总和。

(2) 获得备选节点的状态信息，其中包含节点的资源信息。

(3) 如果备选Pod和节点中已存在Pod的所有容器的需求资源（内存和CPU）的总和，超出了备选节点拥有的资源，则返回 `false`，表明备选节点不适合备选Pod，否则返回 `true`，表明备选节点适合备选Pod。

3) PodSelectorMatches

判断备选节点是否包含备选Pod的标签选择器指定的标签。

(1) 如果Pod没有指定 `spec.nodeSelector` 标签选择器，则返回 `true`。

(2) 否则，获得备选节点的标签信息，判断节点是否包含备选Pod的标签选择器（`spec.nodeSelector`）所指定的标签，如果包含，则

返回true，否则返回false。

4) PodFitsHost

判断备选Pod的spec.nodeName域所指定的节点名称和备选节点的名称是否一致，如果一致，则返回true，否则返回false。

5) CheckNodeLabelPresence

如果用户在配置文件中指定了该策略，则 Scheduler 会通过 RegisterCustomFitPredicate 方法注册该策略。该策略用于判断策略列出的标签在备选节点中存在时，是否选择该备选节点。

(1) 读取备选节点的标签列表信息。

(2) 如果策略配置的标签列表存在于备选节点的标签列表中，且策略配置的presence值为false，则返回false，否则返回true；如果策略配置的标签列表不存在于备选节点的标签列表中，且策略配置的presence值为true，则返回false，否则返回true。

6) CheckServiceAffinity

如果用户在配置文件中指定了该策略，则 Scheduler 会通过 RegisterCustomFitPredicate 方法注册该策略。该策略用于判断备选节点是否包含策略指定的标签，或包含和备选 Pod 在相同 Service 和 Namespace 下的 Pod 所在节点的标签列表。如果存在，则返回true，否则返回false。

7) PodFitsPorts

判断备选Pod所用的端口列表中的端口是否在备选节点中已被占用，如果被占用，则返回false，否则返回true。

Scheduler 中的 优选策略包含：LeastRequestedPriority、CalculateNodeLabelPriority和BalancedResourceAllocation等。每个节点通过优先选择策略时都会算出一个得分，计算各项得分，最终选出得分值最大的节点作为优选的结果（也是调度算法的结果）。

下面是对所有优选策略的详细说明。

1) LeastRequestedPriority

该优选策略用于从备选节点列表中选出资源消耗最小的节点。

(1) 计算出所有备选节点上运行的Pod和备选Pod的CPU占用量totalMilliCPU。

(2) 计算出所有备选节点上运行的Pod和备选Pod的内存占用量totalMemory。

(3) 计算每个节点的得分，计算规则大致如下。

NodeCpuCapacity为节点CPU计算能力；NodeMemoryCapacity为节点内存大小。

```
score=int(((nodeCpuCapacity-totalMilliCPU)*10)/
nodeCpuCapacity+((nodeMemoryCapacity-totalMemory)*10)/
nodeCpuMemory)/2)
```

2) CalculateNodeLabelPriority

如果用户在配置文件中指定了该策略，则scheduler会通过RegisterCustomPriorityFunction方法注册该策略。该策略用于判断策略列出的标签在备选节点中存在时，是否选择该备选节点。如果备选节点的标签在优选策略的标签列表中且优选策略的presence值为true，或者备选节点的标签不在优选策略的标签列表中且优选策略的presence值为false，则备选节点score=10，否则备选节点score=0。

3) BalancedResourceAllocation

该优选策略用于从备选节点列表中选出各项资源使用率最均衡的节点。

(1) 计算出所有备选节点上运行的Pod和备选Pod的CPU占用量totalMilliCPU。

(2) 计算出所有备选节点上运行的Pod和备选Pod的内存占用量totalMemory。

(3) 计算每个节点的得分，计算规则大致如下。

NodeCpuCapacity为节点CPU计算能力；NodeMemoryCapacity为节点内存大小。

```
score= int(10-math.Abs(totalMilliCPU/nodeCpuCapacity-  
totalMemory/ nodeMemoryCapacity)*10)
```

3.4 kubelet运行机制分析

在Kubernetes集群中，在每个Node节点（又称Minion）上都会启动一个kubelet服务进程。该进程用于处理Master节点下发到本节点的任务，管理Pod及Pod中的容器。每个kubelet进程会在API Server上注册节点自身信息，定期向Master节点汇报节点资源的使用情况，并通过cAdvisor监控容器和节点资源。

3.4.1 节点管理

节点通过设置kubelet的启动参数“--register-node”，来决定是否向API Server注册自己。如果该参数的值为true，那么kubelet将试着通过API Server注册自己。在自注册时， kubelet启动时还包含下列参数。

- --api-servers: 告诉kubelet API Server的位置。
- --kubeconfig: 告诉kubelet在哪儿可以找到用于访问API Server的证书。
- --cloud-provider: 告诉kubelet如何从云服务商（IaaS）那里读取到和自己相关的元数据。

当前每个kubelet被授予创建和修改任何节点的权限。但是在实践中，它仅仅创建和修改自己。将来，我们计划限制kubelet的权限，仅允许它修改和创建其所在节点的权限。如果在集群运行过程中遇到集群资源不足的情况，则用户很容易通过添加机器及运用kubelet的自注册模式来实现扩容。

在某些情况下，Kubernetes集群中的某些kubelet没有选择自注册模式，用户需要自己去配置Node的资源信息，同时告知Node上的kubelet API Server的位置。集群管理者能够创建和修改节点信息。如果管理者希望手动创建节点信息，则通过设置kubelet的启动参数“--register-node=false”即可。

kubelet在启动时通过API Server注册节点信息，并定时向API Server发送节点的新消息，API Server在接收到这些信息后，将这些信

息写入etcd。通过kubelet的启动参数“`--node-status-update-frequency`”设置kubelet每隔多少时间向API Server报告节点状态，默认为10秒。

3.4.2 Pod管理

kubelet通过以下几种方式获取自身Node上所要运行的Pod清单。

(1) 文件： kubelet启动参数“--config”指定的配置文件目录下的文件（默认目录为“/etc/kubernetes/manifests/”）。通过 --file-check-frequency设置检查该文件目录的时间间隔，默认为20秒。

(2) HTTP端点（URL）： 通过“--manifest-url”参数设置。通过--http-check-frequency设置检查该HTTP端点数据的时间间隔，默认为20秒。

(3) API Server： kubelet通过API Server监听etcd目录，同步Pod列表。

所有以非API Server方式创建的Pod都叫作Static Pod。kubelet将Static Pod的状态汇报给API Server，API Server为该Static Pod创建一个Mirror Pod和其相匹配。Mirror Pod的状态将真实反映Static Pod的状态。当Static Pod被删除时，与之相对应的Mirror Pod也会被删除。在本章中我们只讨论通过APIServer获得Pod清单的方式。kubelet通过API Server Client使用Watch加List的方式监听“/registry/nodes/\$当前节点的名称”和“/registry/pods”目录，将获取的信息同步到本地缓存中。

kubelet监听etcd，所有针对Pod的操作将会被kubelet监听到。如果发现新的绑定到本节点的Pod，则按照Pod清单的要求创建该Pod。

如果发现本地的Pod被修改，则kubelet会做出相应的修改，比如删除Pod中的某个容器时，则通过Docker Client删除该容器。

如果发现删除本节点的Pod，则删除相应的Pod，并通过Docker Client删除Pod中的容器。

kubelet读取监听到的信息，如果是创建和修改Pod任务，则做如下处理。

- (1) 为该Pod创建一个数据目录。
- (2) 从API Server读取该Pod清单。
- (3) 为该Pod挂载外部卷（External Volume）。
- (4) 下载Pod用到的Secret。

(5) 检查已经运行在节点中的Pod，如果该Pod没有容器或Pause容器（“kubernetes/pause”镜像创建的容器）没有启动，则先停止Pod里所有容器的进程。如果在Pod中有需要删除的容器，则删除这些容器。

(6) 用“kubernetes/pause”镜像为每个Pod创建一个容器。该Pause容器用于接管Pod中所有其他容器的网络。每创建一个新的Pod，kubelet都会先创建一个Pause容器，然后创建其他容器。“kubernetes/pause”镜像大概为200KB，是一个非常小的容器镜像。

- (7) 为Pod中的每个容器做如下处理。

- 为容器计算一个hash值，然后用容器的名字去查询对应Docker容器的hash值。若查找到容器，且两者的hash值不同，则停止Docker中容器的进程，并停止与之关联的Pause容器的进程；若两者相同，则不做任何处理。
- 如果容器被终止了，且容器没有指定的restartPolicy（重启策略），则不做任何处理。
- 调用Docker Client下载容器镜像，调用Docker Client运行容器。

3.4.3 容器健康检查

Pod通过两类探针来检查容器的健康状态。一个是LivenessProbe探针，用于判断容器是否健康，告诉kubelet一个容器什么时候处于不健康的状态。如果LivenessProbe探针探测到容器不健康，则kubelet将删除该容器，并根据容器的重启策略做相应的处理。如果一个容器不包含LivenessProbe探针，那么kubelet认为该容器的LivenessProbe探针返回的值永远是“Success”；另一类是ReadinessProbe探针，用于判断容器是否启动完成，且准备接收请求。如果ReadinessProbe探针检测到失败，则Pod的状态将被修改。Endpoint Controller将从Service的Endpoint中删除包含该容器所在Pod的IP地址的Endpoint条目。

kubelet定期调用容器中的LivenessProbe探针来诊断容器的健康状况。LivenessProbe包含以下三种实现方式。

(1) ExecAction: 在容器内部执行一个命令，如果该命令的退出状态码为0，则表明容器健康。

(2) TCPSocketAction: 通过容器的IP地址和端口号执行TCP检查，如果端口能被访问，则表明容器健康。

(3) HTTPGetAction: 通过容器的IP地址和端口号及路径调用HTTP Get方法，如果响应的状态码大于等于200且小于等于400，则认为容器状态健康。

LivenessProbe 探针包含在 Pod 定义的 `spec.containers.{ 某个容器 }` 中。下面的例子展示了两种 Pod 中容器健康检查的方式：HTTP 检查和容器命令执行检查。下面所列的内容实现了通过容器命令执行检查：

```
livenessProbe:
  exec:
    command:
      - cat
      - /tmp/health
  initialDelaySeconds: 15
  timeoutSeconds: 1
```

kubelet 在容器中执行“cat/tmp/health”命令，如果该命令返回的值为 0，则表明容器处于健康状态，否则表明容器处于不健康状态。

下面所列的内容实现了容器的 HTTP 检查：

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 15
  timeoutSeconds: 1
```

kubelet 发送一个 HTTP 请求到本地主机和端口及指定的路径，来检查容器的健康状况。

3.4.4 cAdvisor资源监控

在Kubernetes集群中如何监控资源的使用情况？

在Kubernetes集群中，应用程序的执行情况可以在不同的级别上监测到，这些级别包括：容器、Pod、Service和整个集群。作为Kubernetes集群的一部分，Kubernetes希望提供给用户详细的各个级别的资源使用信息，这将使用户能够深入地了解应用的执行情况，并找到应用中可能的瓶颈。Heapster项目为Kubernetes提供了一个基本的监控平台，它是集群级别的监控和事件数据集成器（Aggregator）。Heapster作为Pod运行在Kubernetes集群中，和运行在Kubernetes集群中的其他应用相似。Heapster Pod通过kubelet（运行在节点上的Kubernetes代理）发现所有运行在集群中的节点，并查看来自这些节点的资源使用状况信息。kubelet通过cAdvisor获取其所在节点及容器的数据，Heapster通过带着关联标签的Pod分组这些信息，这些数据被推到一个可配置的后端，用于存储和可视化展示。当前支持的后端包括 InfluxDB （ with Grafana for Visualization ） 和 Google Cloud Monitoring。

cAdvisor是一个开源的分析容器资源使用率和性能特性的代理工具。它是因为容器而产生的，因此自然支持Docker容器。在Kubernetes项目中，cAdvisor被集成到Kubernetes代码中。cAdvisor自动查找所有在其所在节点上的容器，自动采集CPU、内存、文件系统和网络使用的统计信息。cAdvisor通过它所在节点机的Root容器，采集并分析该节点机的全面使用情况。

在大部分Kubernetes集群中，cAdvisor通过它所在节点机的4194端口暴露一个简单的UI。图3.8是cAdvisor的一个截图。

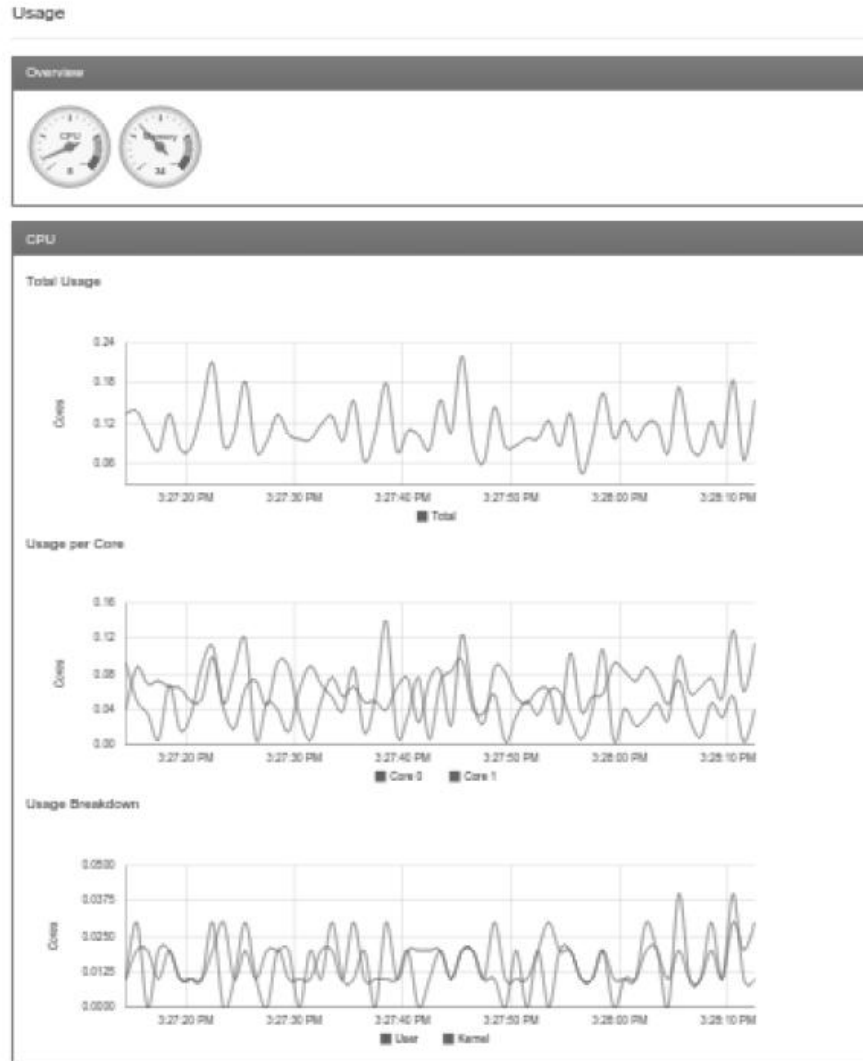


图3.8 cAdvisor的一个UI

kubelet作为连接Kubernetes Master和各节点机之间的桥梁，管理运行在节点机上的Pod和容器。kubelet将每个Pod转换成它的成员容器，同时从cAdvisor获取单独的容器使用统计信息，然后通过该REST API暴露这些聚合后的Pod资源使用的统计信息。

3.5 kube-proxy运行机制分析

我们在前面已经了解到，为了支持集群的水平扩展、高可用性，Kubernetes抽象出了Service的概念。Service是对一组Pod的抽象，它会根据访问策略（如负载均衡策略）来访问这组Pod。

Kubernetes在创建服务时会为服务分配一个虚拟的IP地址，客户端通过访问这个虚拟的IP地址来访问服务，而服务则负责将请求转发到后端的Pod上。这不就是一个反向代理吗？不错，这就是一个反向代理。但是，它和普通的反向代理有一些不同：首先它的IP地址是虚拟的，想从外面访问还需要一些技巧；其次是它的部署和启停是Kubernetes统一自动管理的。

Service在很多情况下只是一个概念，而真正将Service的作用落实的是背后的kube-proxy服务进程。只有理解了kube-proxy的原理和机制，我们才能真正理解Service背后的实现逻辑。

在Kubernetes集群的每个Node上都会运行一个kube-proxy服务进程，这个进程可以看作Service的透明代理兼负载均衡器，其核心功能是将到某个Service的访问请求转发到后端的多个Pod实例上。对每一个TCP类型的Kubernetes Service，kube-proxy都会在本Node上建立一个SocketServer来负责接收请求，然后均匀发送到后端某个Pod的端口上，这个过程默认采用Round Robin负载均衡算法。另外，Kubernetes也提供通过修改Service的service.spec.sessionAffinity参数的值来实现会

话保持特性的定向转发，如果设置的值为“ClientIP”，则将来自同一个ClientIP的请求都转发到同一个后端Pod上。

此外，Service的Cluster IP与NodePort等概念是kube-proxy服务通过Iptables的NAT转换实现的，kube-proxy在运行过程中动态创建与Service相关的Iptables规则，这些规则实现了Cluster IP及NodePort的请求流量重定向到kube-proxy进程上对应服务的代理端口的功能。由于Iptables机制针对的是本地的kube-proxy端口，所以每个Node上都要运行kube-proxy组件，这样一来，在Kubernetes集群内部，我们可以在任意Node上发起对Service的访问请求。

综上所述，由于kube-proxy的作用，在Service的调用过程中客户端无须关心后端有几个Pod，中间过程的通信、负载均衡及故障恢复都是透明的，如图3.9所示。

访问Service的请求，不论是用Cluster IP+TargetPort的方式，还是用节点机IP+NodePort的方式，都被节点机的Iptables规则重定向到kube-proxy监听Service服务代理端口。kube-proxy接收到Service的访问请求后，会如何选择后端的Pod呢？

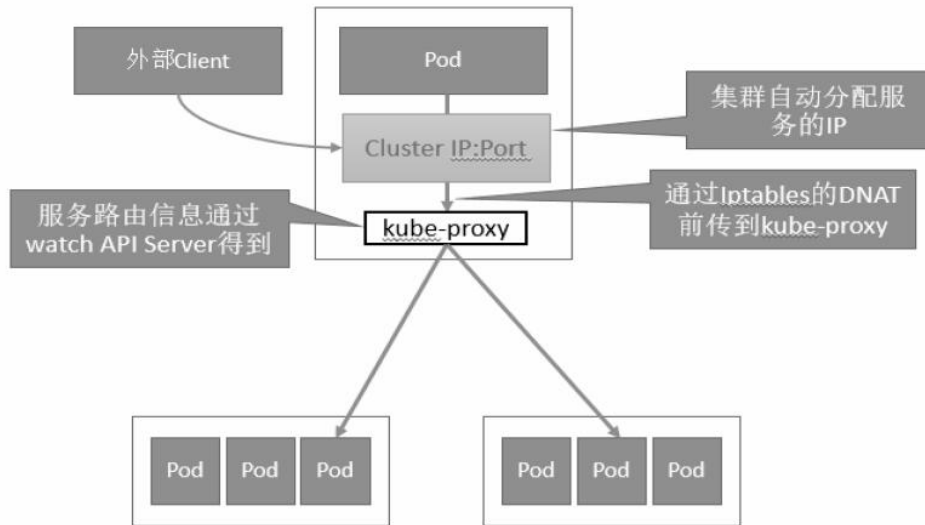


图3.9 Service的负载均衡转发规则

首先，目前kube-proxy的负载均衡器只支持RoundRobin算法。RoundRobin算法按照成员列表逐个选取成员，如果一轮循环完，便从头开始下一轮，如此循环往复。kube-proxy的负载均衡器在RoundRobin算法的基础上还支持Session保持。如果Service在定义中指定了Session保持，则kube-proxy接收请求时会从本地内存中查找是否存在来自该请求IP的affinityState对象，如果存在该对象，且Session没有超时，则kube-proxy将请求转向该affinityState所指向的后端Pod。如果本地存在没有来自该请求IP的affinityState对象，则按照RoundRobin算法为该请求挑选一个Endpoint，并创建一个affinityState对象，记录请求的IP和指向的Endpoint。后面的请求就会“黏连”到这个创建好的affinityState对象上，这就实现了客户端IP会话保持的功能。

接下来我们深入分析kube-proxy的实现细节。

kube-proxy通过查询和监听API Server中Service与Endpoints的变化，为每个Service都建立了一个“服务代理对象”，并自动同步。服务

代理对象是**kube-proxy**程序内部的一种数据结构，它包括一个用于监听此服务请求的**SocketServer**，**SocketServer**的端口是随机选择的一个本地空闲端口。此外，**kube-proxy**内部也创建了一个负载均衡器——**LoadBalancer**，**LoadBalancer**上保存了**Service**到对应的后端**Endpoint**列表的动态转发路由表，而具体的路由选择则取决于**Round Robin**负载均衡算法及**Service**的**Session**会话保持（**SessionAffinity**）这两个特性。

针对发生变化的**Service**列表，**kube-proxy**会逐个处理。下面是具体的处理流程。

（1）如果该**Service**没有设置集群IP（**ClusterIP**），则不做任何处理，否则，获取该**Service**的所有端口定义列表（**spec.ports**域）。

（2）逐个读取服务端口定义列表中的端口信息，根据端口名称、**Service**名称和**Namespace**判断本地是否已经存在对应的服务代理对象，如果不存在则新建；如果存在并且**Service**端口被修改过，则先删除**Iptables**中和该**Service**端口相关的规则，关闭服务代理对象，然后走新建流程，即为该**Service**端口分配服务代理对象并为该**Service**创建相关的**Iptables**规则。

（3）更新负载均衡器组件中对应**Service**的转发地址列表，对于新建的**Service**，确定转发时的会话保持策略。

（4）对于已经删除的**Service**则进行清理。

而针对**Endpoint**的变化，**kube-proxy**会自动更新负载均衡器中对应**Service**的转发地址列表。

下面讲解**kube-proxy**针对**Iptables**所做的一些细节操作。

kube-proxy在启动时和监听到Service或Endpoint的变化后，会在本机Iptables的NAT表中添加4条规则链。

(1) KUBE-PORTALS-CONTAINER：从容器中通过Service Cluster IP和端口号访问Service的请求。

(2) KUBE-PORTALS-HOST：从主机中通过Service Cluster IP和端口号访问Service的请求。

(3) KUBE-NODEPORT-CONTAINER：从容器中通过Service的NodePort端口号访问Service的请求。

(4) KUBE-NODEPORT-HOST：从主机中通过Service的NodePort端口号访问Service的请求。

此外，kube-proxy在Iptables中为每个Service创建由Cluster IP+Service端口到kube-proxy所在主机IP+Service代理服务所监听的端口的转发规则。转发规则的包匹配规则部分（CRETIRIA）如下所示：

```
-m comment --comment $SERVICESTRING -p $PROTOCOL -m $PROTOCOL --dport $DESTPORT -d $DESTIP
```

其中，“-m comment--comment”表示匹配规则使用Iptables的显式扩展的注释功能；“\$SERVICESTRING”为注释的内容；“-p\$PROTOCOL-m\$PROTOCOL--dport\$DESTPORT-d\$DESTIP”表示协议为“\$PROTOCOL”且目标地址和端口为“\$DESTIP”和“\$DESTPORT”的包，其中，“\$PROTOCOL”可以为TCP或UDP，“\$DESTIP”和“\$DESTPORT”为Service的Cluster IP和TargetPort。

对于转发规则的跳转部分（-j部分），如果请求来自本地容器，且Service代理服务监听的是所有的接口（例如IPv4的地址为0.0.0.0），则跳转部分如下所示：

```
-j REDIRECT --to-ports $proxyPort
```

其表示该规则的功能是实现数据包的端口重定向，重定向到\$proxyPort端口（Service代理服务监听的端口）；否则，跳转部分如下所示：

```
-j DNAT --to-destination proxyIP:proxyPort
```

表示该规则的功能是实现数据包转发，数据包的目的地址变为“proxyIP: proxyPort”（即Service代理服务所在的IP地址和端口，这些地址和端口都会被替换成实际的地址和端口）。

如果Service类型为NodePort，则kube-proxy在Iptables中除了添加上面提及的规则，还会为每个Service创建由NodePort端口到kube-proxy所在主机IP+Service代理服务所监听的端口的转发规则。转发规则的包匹配规则部分（CRETIRIA）如下所示：

```
-m comment --comment $SERVICESTRING -p $PROTOCOL -m $PROTOCOL --dport $NODEPORT
```

上面所列的内容用于匹配目的端口为“\$NODEPORT”的包。

转发规则的跳转部分（-j部分）和前面提及的跳转规则一致。

最后，我们以本书第2章的Hello World为例，看看kube-proxy为redis-master服务所生成的Iptables转发规则：

```
$ iptables-save | grep redis-master

-A KUBE-PORTALS-CONTAINER -d 10.254.208.57/32 -p tcp -
m comment --comment "default/redis-master:" -m tcp --dport
6379 -j REDIRECT --to-ports 42872

-A KUBE-PORTALS-HOST -d 10.254.208.57/32 -p tcp -m
comment --comment "default/redis-master:" -m tcp --dport
6379 -j DNAT --to-destination 192.168.1.130:42872
```

可以看到，对“redis-master”Service的6379端口的访问将会被转发到物理机的42872端口上。而42872端口就是kube-proxy为这个Service打开的随机本地端口。

最后，给出本节的一个总结性的示意图，如图3.10所示。

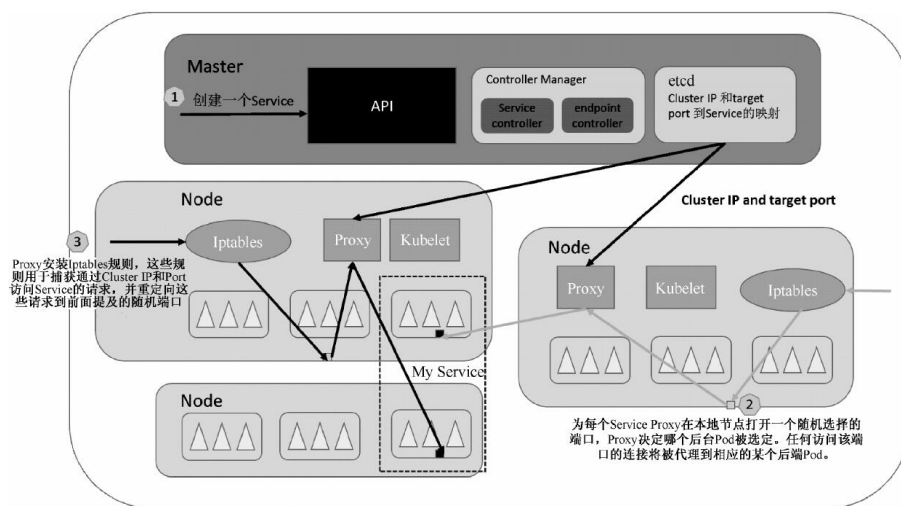


图3.10 kube-proxy工作原理示意图

3.6 深入分析集群安全机制

Kubernetes通过一系列机制来实现集群的安全控制，其中包括API Server的认证授权、准入控制机制及保护敏感信息的Secret机制等。集群的安全性必须考虑如下几个目标。

- (1) 保证容器与其所在的宿主机的隔离。
- (2) 限制容器给基础设施及其他容器带来消极影响的能力。
- (3) 最小权限原则——合理限制所有组件的权限，确保组件只执行它被授权的行为，通过限制单个组件的能力来限制它所能到达的权限范围。
- (4) 明确组件间边界的划分。
- (5) 划分普通用户和管理员的角色。
- (6) 在必要的时候允许将管理员权限赋给普通用户。
- (7) 允许拥有“Secret”数据（Keys、Certs、Passwords）的应用在集群中运行。

下面分别从Authentication、Authorization、Admission Control、Secret和Service Account等方面来说明集群的安全机制。

3.6.1 API Server认证

我们知道，Kubernetes集群中所有资源的访问和变更都是通过Kubernetes API Server的REST API来实现的，所以集群安全的关键点就在于如何识别并认证客户端身份（Authentication），以及随后访问权限的授权（Authorization）这两个关键问题，本节我们讲解前一个问题。

我们知道，Kubernetes集群提供了3种级别的客户端身份认证方式。

- 最严格的HTTPS证书认证：基于CA根证书签名的双向数字证书认证方式。
- HTTP Token认证：通过一个Token来识别合法用户。
- HTTP Base认证：通过用户名+密码的方式认证。

首先，我们说说HTTPS证书认证的原理。

这里需要有一个CA证书，我们知道CA是PKI系统中通信双方都信任的实体，被称为可信第三方（Trusted Third Party，TTP）。CA作为可信第三方的重要条件之一就是CA的行为具有非否认性。作为第三方而不是简单的上级，就必须能让信任者有追究自己责任的能力。CA通过证书证实他人的公钥信息，证书上有CA的签名。用户如果因为信任证书而有了损失，则证书可以作为有效的证据用于追究CA的法律责任。正是因为CA承担责任的承诺，所以CA也被称为可信第三方。在很多情况下，CA与用户是相互独立的实体，CA作为服务提供方，有

可能因为服务质量问题（例如，发布的公钥数据有错误）而给用户带来损失。在证书中绑定了公钥数据和相应私钥拥有者的身份信息，并带有CA的数字签名；证书中也包含了CA的名称，以便于依赖方找到CA的公钥，验证证书上的数字签名。

CA认证涉及诸多概念，比如根证书、自签名证书、密钥、私钥、加密算法及HTTPS等，本书大致讲述SSL协议的流程，有助于对CA认证和Kubernetes CA认证的配置过程的理解。

如图3.11所示，SSL双向认证大概包含下面几个步骤。

(1) HTTPS通信双方的服务器端向CA机构申请证书，CA机构是可信的第三方机构，它可以是一个公认的权威的企业，也可以是企业自身。企业内部系统一般都用企业自身的认证系统。CA机构下发根证书、服务端证书及私钥给申请者。

(2) HTTPS通信双方的客户端向CA机构申请证书，CA机构下发根证书、客户端证书及私钥给申请者。

(3) 客户端向服务器端发起请求，服务端下发服务端证书给客户端。客户端接收到证书后，通过私钥解密证书，并利用服务器端证书中的公钥认证证书信息比较证书里的消息，例如域名和公钥与服务器刚刚发送的相关消息是否一致，如果一致，则客户端认可这个服务器的合法身份。

(4) 客户端发送客户端证书给服务器端，服务端接收到证书后，通过私钥解密证书，获得客户端证书公钥，并用该公钥认证证书信息，确认客户端是否合法。

(5) 客户端通过随机密钥加密信息，并发送加密后的信息给服务端。服务器端和客户端协商好加密方案后，客户端会产生一个随机的密钥，客户端通过协商好的加密方案，加密该随机密钥，并发送该随机密钥到服务器端。服务器端接收这个密钥后，双方通信的所有内容都通过该随机密钥加密。

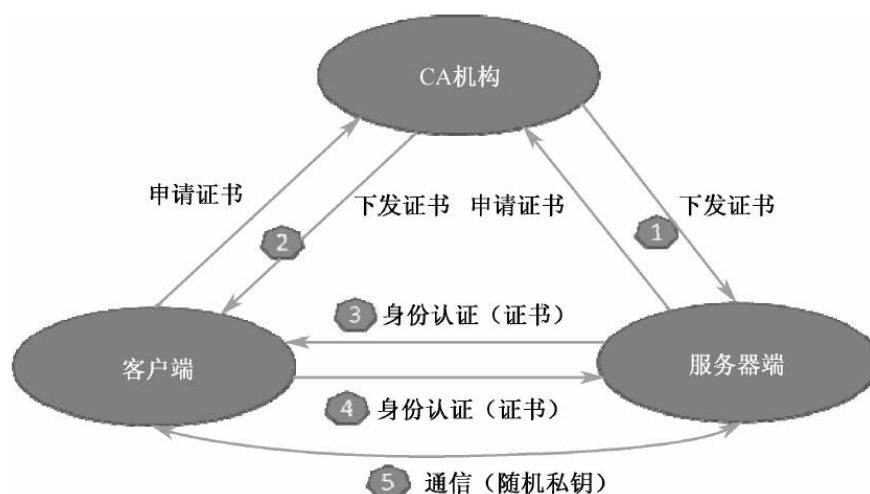


图3.11 CA认证流程

如上所述是双向认证SSL协议的具体通信过程，这种情况要求服务器和用户双方都有证书。单向认证SSL协议不需要客户拥有CA证书，对于上面的步骤，只需将服务器端验证客户证书的过程去掉，以及在协商对称密码方案 and 对称会话密钥时，服务器发送给客户的是没有加过密的（这并不影响SSL过程的安全性）密码方案。

其次，我们来看看HTTP Token的认证原理。

HTTP Token的认证是用一个很长的特殊编码方式的并且难以被模仿的字符串——Token来表明客户身份的一种方式。在通常情况下，Token是一个很复杂的字符串，比如我们用私钥签名一个字符串后的数

据就可以当作一个Token。此外，每个Token对应一个用户名，存储在API Server能访问的一个文件中。当客户端发起API调用请求时，需要在HTTP Header里放入Token，这样一来，API Server就能识别合法用户和非法用户了。

最后，我们说说HTTP Base认证。

我们知道，HTTP协议是无状态的，浏览器和Web服务器之间可以通过Cookie来进行身份识别。桌面应用程序（比如新浪桌面客户端、SkyDrive客户端、命令行程序）一般不会使用Cookie，那么它们与Web服务器之间是如何进行身份识别的呢？这就用到了HTTP Base认证，这种认证方式是把“用户名+冒号+密码”用BASE64算法进行编码后的字符串放在HTTP Request中的Header Authorization域里发送给服务端，服务端收到后进行解码，获取用户名及密码，然后进行用户身份的鉴权过程。

3.6.2 API Server授权

对合法用户进行授权（**Authorization**）并且随后在用户访问时进行鉴权，是权限与安全系统的重要一环。简单地说，授权就是授予不同的用户不同的访问权限，API Server目前支持以下几种授权策略（通过API Server的启动参数“--authorization_mode”设置）。

- AlwaysDeny。
- AlwaysAllow。
- ABAC。

其中，AlwaysDeny表示拒绝所有的请求，该配置一般用于测试；AlwaysAllow表示接收所有的请求，如果集群不需要授权流程，则可以采用该策略，这也是Kubernetes的默认配置；ABAC（Attribute-Based Access Control）为基于属性的访问控制，表示使用用户配置的授权规则去匹配用户的请求。

为了简化授权的复杂度，对于ABAC模式的授权策略，Kubernetes仅有下面四个基本属性。

- 用户名（代表一个已经被认证的用户的字符型用户名）。
- 是否是只读请求（REST的GET操作是只读的）。
- 被访问的是哪一类资源，例如访问Pod资源/api/v1/namespaces/default/pods。
- 被访问对象所属的Namespace。

当我们为API Server启用ABAC模式时，需要指定授权策略文件的路径和名字（`--authorization_policy_file=SOME_FILENAME`），授权策略文件里的每一行都是一个Map类型的JSON对象，被称为“访问策略对象”，我们可以通过设置“访问策略对象”中的如下属性来确定具体的授权行为。

- **user**（用户名）：为字符串类型，该字符串类型的用户名来源于Token文件或基本认证文件中的用户名字段的值。
- **readonly**（只读标识）：为布尔类型，当它的值为true时，表明该策略允许GET请求通过。
- **resource**（资源）：为字符串类型，来自于URL的资源，例如“Pods”。
- **namespace**（命名空间）：为字符串类型，表明该策略允许访问某个Namespace的资源。

例如，我们要实现如下访问控制。

- （1）允许用户alice做任何事情
- （2）kubelet只能访问Pod的只读API。
- （3）kubelet能读和写Event对象。
- （4）用户bob只能访问myNamespace中的Pod的只读API。

则满足上述要求的授权策略文件的内容写法如下：

```
{"user": "alice"}  
  
{"user": "kubelet", "resource": "pods", "readonly":
```

```
true}
    {"user":"kubelet", "resource": "events"}
    {"user":"bob", "resource": "pods", "readonly": true,
"ns": " myNamespace "}
```

当客户端发起API Server调用时，API Server内部要先进行用户认证，接下来执行用户鉴权流程，鉴权流程通过之前提到的“授权策略”来决定一个API调用是否合法。当API Server接收到请求后，会读取该请求中的数据，生成一个“访问策略对象”，如果该请求中不带某些属性（如Namespace），则这些属性的值将根据属性类型的不同，设置不同的默认值（例如为字符串类型的属性设置一个空字符串；为布尔类型的属性设置false；为数值类型的属性设置0）。然后用这个“访问策略对象”和授权策略文件中的所有“访问策略对象”逐条匹配，如果至少有一个策略对象被匹配上，则该请求将被鉴权通过，否则终止API调用流程，并返回客户端错误调用码。

3.6.3 Admission Control准入控制

突破了之前所说的认证和鉴权两道关口之后，客户端的调用请求就能够得到API Server的真正响应了吗？答案是：不能！这个请求还需要通过Admission Control所控制的一个“准入控制链”的层层考验，官方标准的“关卡”有近十个之多，而且能自定义扩展！笔者忽然在想，如果在幼儿园的时候，老师就告诉我们长大后还要读小学，参加中考、高考、公司面试、职称考试，等等，我们还会天天去幼儿园吗？

Admission Control配备有一个“准入控制器”的列表，发送给API Server的任何请求都需要通过列表中每个准入控制器的检查，检查不通过，则API Server拒绝此调用请求。此外，准入控制器还能够修改请求参数以完成一些自动化的任务，比如ServiceAccount这个控制器。当前可配置的准入控制器如下。

- AlwaysAdmit: 允许所有请求。
- AlwaysPullImages: 在启动容器之前总是去下载镜像，相当于在每个容器的配置项imagePullPolicy=Always。
- AlwaysDeny: 禁止所有请求，一般用于测试。
- DenyExecOnPrivileged: 它会拦截所有想在Privileged Container上执行命令的请求。如果你的集群支持Privileged Container，你又希望限制用户在这些Privileged Container上执行命令，那么强烈推荐你使用它。
- ServiceAccount: 这个plug-in将serviceAccounts实现了自动化，默认启用，如果你想要使用ServiceAccount对象，那么强烈推荐你使

用它，后面讲述ServiceAccount的章节会详细说明其作用。

- **SecurityContextDeny**: 这个插件将使用了SecurityContext的Pod中定义的选项全部失效。SecurityContext在Container中定义了操作系统级别的安全设定（uid、gid、capabilities、SELinux等）。
- **ResourceQuota**: 用于配额管理目的，作用于Namespace上，它会观察所有的请求，确保在namespace上的配额不会超标。推荐在Admission Control参数列表中这个插件排最后一个。
- **LimitRanger**: 用于配额管理，作用于Pod与Container上，确保Pod与Container上的配额不会超标。
- **NamespaceExists**（已过时）: 对所有请求校验namespace是否已存在，如果不存在则拒绝请求。已合并至NamespaceLifecycle。
- **NamespaceAutoProvision**（已过时）: 对所有请求校验namespace，如果不存在则自动创建该namespace，推荐使用NamespaceLifecycle。
- **NamespaceLifecycle**: 如果尝试在一个不存在的namespace中创建资源对象，则该创建请求将被拒绝。当删除一个namespace时，系统将会删除该namespace中的所有对象，包括Pod、Service等。

在API Server上设置--admission-control参数，即可定制我们需要的准入控制链，如果启用多种准入控制选项，则建议的设置（含加载顺序）如下：

```
--admission-control=NamespaceLifecycle,LimitRanger,SecurityContextDeny,ServiceAccount,ResourceQuota
```

大部分准入控制器都比较容易理解，我们接下来着重介绍SecurityContextDeny、ResourceQuota及LimitRanger这三个准入控制器。

1) SecurityContextDeny

SecurityContext是运用于容器的操作系统安全设置（uid、gid、capabilities、SELinux role等）。Admission Control的SecurityContextDeny插件的作用是，禁止创建设置了SecurityContext的Pod，例如包含下面这些配置项的Pod：

```
spec.containers.securityContext.seLinuxOptions
spec.containers.securityContext.runAsUser
```

2) ResourceQuota

准入控制器ResourceQuota不仅能够限制某个Namespace中创建资源的数量，而且能够限制某个Namespace中被Pod所请求的资源总量。该准入控制器和资源对象ResourceQuota一起实现了资源配额管理。

3) LimitRanger

准入控制器LimitRanger的作用类似于上面的ResourceQuota控制器，针对Namespace资源的每个个体（Pod与Container等）的资源配额。该插件和资源对象LimitRange一起实现资源限制管理。

3.6.4 Service Account

Service Account也是一种账号，但它并不是给Kubernetes的集群的用户（系统管理员、运维人员、租户用户等）使用的，而是给运行在Pod里的进程用的，它为Pod里的进程提供必要的身份证明。

在继续学习之前，请回忆一下本章前面所说的API Server的认证一节。

我们知道，正常情况下，为了确保Kubernetes集群的安全，API Server都会对客户端进行身份认证，认证失败的客户端无法进行API调用。此外，Pod中访问Kubernetes API Server服务的时候，是以Service方式访问服务名为kubernetes的这个服务的，而kubernetes服务又只在HTTPS安全端口443上提供服务，那么如何进行身份认证呢？这的确是个谜，因为Kubernetes的官方文档并没有清楚说明这个问题。

通过查看官方源码，我们发现这是在用一种类似HTTP Token的新的认证方式——Service Account Auth，Pod中的客户端调用kubernetesAPI的时候，在HTTP Header中传递了一个Token字符串，这类似于之前提到的HTTP Token认证方式，但又有以下几个不同点。

- 这个Token的内容来自Pod里指定路径下的一个文件（/run/secrets/kubernetes.io/serviceaccount/token），这种Token是动态生成的，确切地说，是由Kubernetes Controller进程用API Server的私钥（--service-account-private-key-file指定的私钥）签名生成的一个JWT Secret。

- 官方提供的客户端REST框架代码里，通过HTTPS方式与API Server建立连接后，会用Pod里指定路径下的一个CA证书（`/run/secrets/kubernetes.io/serviceaccount/ca.crt`）验证API Server发来的证书，验证是否是被CA证书签名的合法证书。
- API Server收到这个Token以后，采用自己的私钥（实际是使用参数`service-account-key-file`指定的私钥，如果此参数没有设置，则默认采用`tls-private-key-file`指定的参数，即自己的私钥）对Token进行合法性验证。

明白了认证原理，我们接下来继续分析上面认证过程中所涉及的Pod中的以下三个文件。

- `/run/secrets/kubernetes.io/serviceaccount/token`。
- `/run/secrets/kubernetes.io/serviceaccount/ca.crt`。
- `/run/secrets/kubernetes.io/serviceaccount/namespace`（客户端采用这里指定的namespace作为参数调用Kubernetes API）。

这三个文件由于参与到Pod进程与API Server认证的过程中，起到了类似Secret（私密凭据）的作用，所以它们被称为Kubernetes Secret对象。Secret从属于Service Account资源对象，属于Service Account的一部分，一个Service Account对象里面可以包括多个不同的Secret对象，分别用于不同目的认证活动。

下面我们通过运行一些命令来加深我们对Service Account与Secret的直观认识。

首先，查看系统中的Service Account对象，我们看到有一个名为default的Service Account对象，包含一个名为default-token-77oyg的

Secret，这个Secret同时是“Mountable secrets”，表明它是需要被Mount到Pod上的：

```
# kubectl describe serviceaccounts
Name:                default
Namespace: default
Labels:              <none>
Image pull secrets:  <none>
Mountable secrets:   default-token-77oyg
Tokens:              default-token-77oyg
```

接下来，我们看看default-token-77oyg都有什么内容：

```
# kubectl describe secrets default-token-77oyg
Name:                default-token-77oyg
Namespace: default
Labels:              <none>
Annotations:         kubernetes.io/service-account.name=default
                    kubernetes.io/service-account.uid=3e5b99c0-432c-11e6-b45c-000c29dc2102
Type:                kubernetes.io/service-account-token

Data
====
```

token:

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNib3VudC9uYW1lc3BhY2UiOiJkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNib3VudC9zZWNyZXQubmFtZSI6ImRlZmF1bHQtdG9rZW4tNzdveWciLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoizGVmYXVsdCIsImt1YmVybmV0ZXMuaW8vc2VydmVjZWJjY291bnQvc2VydmVjZS1hY2NvdW50LnVpZCI6IjNlNWl5OWMwLTQzMmMtMTF1Ni1iNDVjLTAwMGMyOWRjMjEwMiIsInN1YiI6InN5c3RlbTpzZXJ2aWNlYWNib3VudDpkZWZhdWx0OmRlZmF1bHQifQ.MFsBrYmTLMB55X3UGf0_pADP6FSsQgHb0SxGJtTsJnY-
```

```
ze2vFc8Qd07bVdmQfFbnkHgLWht1KIpr_EyvJTRP538uovgcA_QGN9yIMEdqIfQC2wfnLFuk10a80dSH4uzayBb50yI7gJWXWbXn6u0wAGMneiTKtCvzGfR4q-
```

```
p19Jjh5qNPiUdJ0NhjsJJSAc1hdNK40XtOgMHdNNyPEmPgk60w2cM7DRb6ifiS0s05cTeLYv1TpIBMvcQy4sYedCEL2cJ20BwcSo4-
```

```
1Dev9rdxr50dtgCvo60xbPF7RcWwjgUMLY03YCi07WmQNdmxWHJkwvBtkWZhhdvuFCpHewANA
```

```
ca.crt: 1115 bytes
```

```
namespace: 7 bytes
```

从上面的输出信息中我们看到，**default-token-77oyg**包括三个数据项，分别是**token**、**ca.crt**、**namespace**。联想到“Mountable secrets”的标记，以及之前看到的Pod中的三个文件的文件名，你可能恍然大悟，原来是这么一回事：每个Namespace下有一个名为**default**的默认的ServiceAccount对象，这个ServiceAccount里面有一个名为**Tokens**的可以当作Volume一样被Mount到Pod里的Secret，当Pod启动时，这个

Secret会自动被Mount到Pod的指定目录下，用来协助完成Pod中的进程访问API Server时的身份鉴权过程。

如图3.12所示，一个Service Account可以包括多个Secret对象。

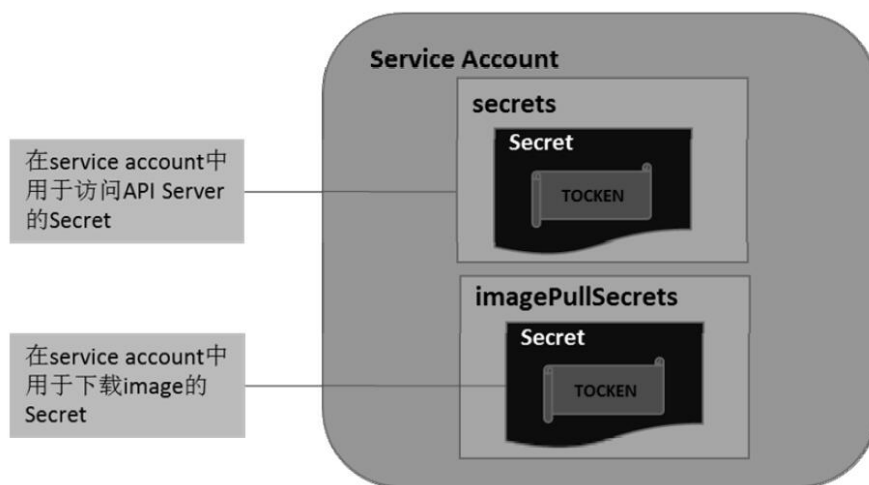


图3.12 Service Account中的Secret

(1) 名为Tokens的Secret用于访问API Server的Secret，也被称为Service Account Secret。

(2) 名为Image pull secrets的Secret用于下载容器镜像时的认证过程，通常镜像库运行在Insecure模式下，所以这个Secret为空。

(3) 用户自定义的其他Secret，用于用户的进程。

如果一个Pod在定义时没有指定spec.serviceAccountName属性，则系统会自动为其赋值为“default”，即大家都使用同一个Namespace下默认的Service Account。如果某个Pod需要使用非default的Service Account，则需要在定义时指定：

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: mycontainer
      image: nginx:v1
    serviceAccountName: myserviceaccount
```

Kubernetes之所以要创建两套独立的账号系统，原因如下。

- User账号是给人用的，Service Account是给Pod里的进程使用的，面向的对象不同。
- User账号是全局性的，Service Account则属于某个具体的Namespace。
- 通常来说，User账号是与后端的用户数据库同步的，创建一个新用户通常要走一套复杂的业务流程才能实现，Service Account的创建则需要极轻量级的实现方式，集群管理员可以很容易为某些特定任务创建一个Service Account。
- 对于这两种不同的账号，其审计要求通常不同。
- 对于一个复杂的系统来说，多个组件通常拥有各种账号的配置信息，Service Account是Namespace隔离的，可以针对组件进行一对一的定义，同时具备很好的“便携性”。

接下来，我们深入分析Service Account与Secret相关的一些运行机制。

前面的Controller Manager原理分析一节中，我们知道Controller manager创建了ServiceAccount Controller与Token Controller两个安全相关的控制器。其中ServiceAccount Controller一直监听Service Account和Namespace的事件，如果一个Namespace中没有default Service Account，那么ServiceAccount Controller就会为该Namespace创建一个默认（default）的Service Account，这就是我们之前看到每个Namespace下都有一个名为default的Service Account的原因了。

如果Controller manager进程在启动时指定了API Server私钥（service-account-private-key-file参数），那么Controller manager会创建Token Controller。Token Controller也监听Service Account的事件，如果发现新创建的Service Account里没有对应的Service Account Secret，则会用API Server私钥创建一个Token（JWT Token），并用该Token、CA证书及Namespace名称等三个信息产生一个新的Secret对象，然后放入刚才的Service Account中；如果监听到的事件是删除Service Account事件，则自动删除与该Service Account相关的所有Secret。此外，Token Controller对象同时监听Secret的创建、修改和删除事件，并根据事件的不同做不同的处理。

当我们在API Server的鉴权过程中启用了Service Account类型的准入控制器，即在kube-apiserver启动参数中包括下面的内容时：

```
--admission_control=ServiceAccount
```

则针对Pod新增或修改的请求，Service Account准入控制器会验证Pod里的Service Account是否合法。

(1) 如果spec.serviceAccount域没有被设置，则Kubernetes默认为其指定名字为default的Service account。

(2) 如果Pod的spec.serviceAccount域指定了default以外的Service Account，而该Service Account没有事先被创建，则该Pod操作失败。

(3) 如果在Pod中没有指定“ImagePullSecrets”，那么这个spec.serviceAccount域指定的Service Account的“ImagePullSecrets”会被加入该Pod中。

(4) 给Pod添加一个特殊的Volume，在该Volume中包含Service Account Secret中的Token，并将Volume挂载到Pod中所有容器的指定目录下（/var/run/secrets/kubernetes.io/serviceaccount）。

综上所述，Service Account的正常工作离不开以下几个控制器。

(1) Admission Controller。

(2) Token Controller。

(3) ServiceAccount Controller。

3.6.5 Secret 私密凭据

上一节我们提到Secret对象，Secret的主要作用是保管私密数据，比如密码、OAuth Tokens、SSH Keys等信息。将这些私密信息放在Secret对象中比直接放在Pod或Docker Image中更安全，也更便于使用和分发。

下面的例子用于创建一个Secret:

```
secrets.yaml:
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  password: dmFsdWUtMg0K
  username: dmFsdWUtMQ0K

# kubectl create -f secrets.yaml
```

在上面的例子中，data域的各子域的值必须为BASE64编码值，其中password域和username域BASE64编码前的值分别为“value-1”和“value-2”。

一旦Secret被创建，则可以通过下面的三种方式使用它。

(1) 在创建Pod时，通过为Pod指定Service Account来自动使用该Secret。

(2) 通过挂载该Secret到Pod来使用它。

(3) Docker 镜像下载时使用，通过指定Pod的spec.ImagePullSecrets来引用它。

第1种使用方式主要用在API Server鉴权方面，之前我们提到过。下面的例子展示了第2种使用方式：将一个Secret通过挂载的方式添加到Pod的Volume中。

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
  namespace: myns
spec:
  containers:
  - name: mycontainer
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
```



```
- name: foo
  secret:
    secretName: mysecret
```

其结果如图3.13所示。

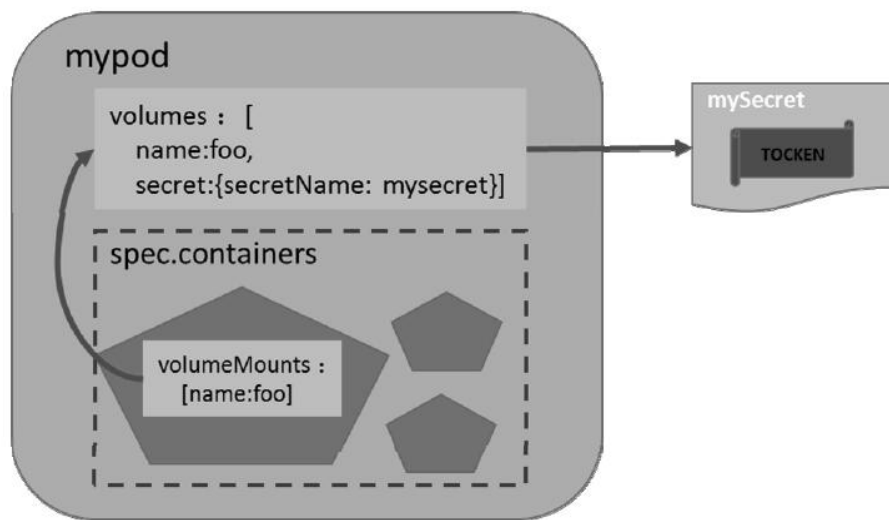


图3.13 挂载Secret到Pod

第3种使用方式的使用流程如下。

(1) 执行login命令，登录私有Registry:

```
# docker login localhost:5000
```

输入用户名和密码，如果是第1次登录系统，则会创建新用户，相关信息会写入~/.dockercfg文件中。

(2) 用BASE64编码dockercfg的内容:

```
# cat ~/.dockercfg | base64
```

(3) 将上一步命令的输出结果作为Secret的“data.dockercfg”域的内容，由此来创建一个Secret:

```
image-pull-secret.yaml:
apiVersion: v1
kind: Secret
metadata:
  name: myregistrykey
data:
  .dockercfg:
    eyAiaHR0cHM6Ly9pbmRleC5kb2NrZXIuaW8vdjEvIjogeyAiYXV0aCI6ICJ
    abUZyWlhCaGMzTjNiM0prTVRJSyIsICJlbWFPbCI6ICJqZG9lQGV4YW1wbG
    UuY29tIiB9IH0K
  type: kubernetes.io/dockercfg

# kubectl create -f image-pull-secret.yaml
```

(4) 在创建Pod时引用该Secret:

```
pods.yaml:
apiVersion: v1
kind: Pod
metadata:
  name: mypod2
spec:
```

```
containers:
  - name: foo
    image: janedoe/awesomeapp:v1
imagePullSecrets:
  - name: myregistrykey
```

```
$ kubectl create -f pods.yaml
```

其结果如图3.14所示。

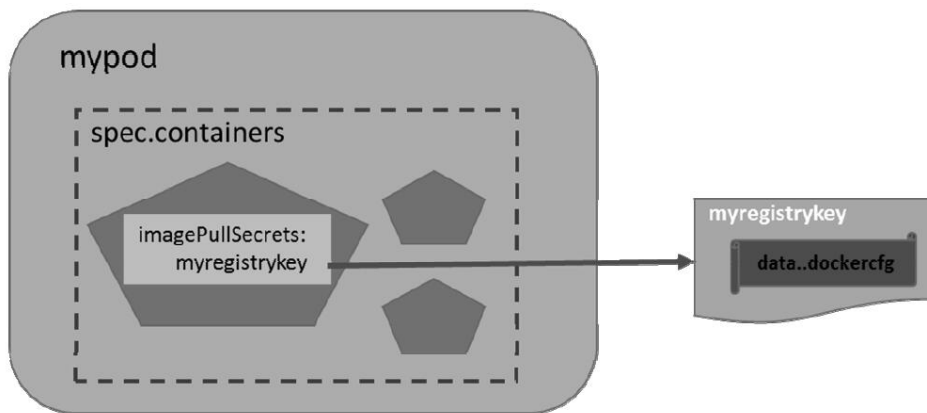


图3.14 imagePullSecret引用Secret

每个单独的Secret大小不能超过1MB，Kubernetes不鼓励创建大尺寸的Secret，因为如果使用大尺寸的Secret，则将大量占用API Server和kubelet的内存。当然，创建许多小的Secret也能耗尽API Server和kubelet的内存。

在使用Mount方式挂载Secret时，Container中Secret的“data”域的各个域的Key值作为目录中的文件，Value值被BASE64编码后存储在相应的文件中。前面的例子中创建的Secret，被挂载到一个叫作mycontainer

的Container中，在该Container中可通过相应的查询命令查看所生成的文件和文件中的内容，如下所示：

```
$ ls /etc/foo/  
username  
password  
$ cat /etc/foo/username  
value-1  
$ cat /etc/foo/password  
value-2
```

通过上面的例子可以得出如下结论：我们可以通过Secret保管其他系统的敏感信息（比如数据库的用户名和密码），并以Mount的方式将Secret挂载到Container中，然后通过访问目录中的文件的方式获取该敏感信息。当Pod被API Server创建时，API Server不会校验该Pod引用的Secret是否存在。一旦这个Pod被调度，则kubelet将试着获取Secret的值。如果Secret不存在或暂时无法连接到API Server，则kubelet将按一定的时间间隔定期重试获取该Secret，并发送一个Event来解释Pod没有启动的原因。一旦Secret被Pod获取，则kubelet将创建并Mount包含Secret的Volume。只有所有Volume被Mount后，Pod中的Container才会被启动。在kubelet启动Pod中的Container后，Container中的和Secret相关的Volume将不会被改变，即使Secret本身被修改了。为了使用更新后的Secret，必须删除旧的Pod，并重新创建一个新的Pod，因此更新Secret的流程和部署一个新的Image是一样的。

3.7 网络原理

关于Kubernetes网络，我们通常有这些问题需要回答，如图3.15所示。



图3.15 Kubernetes的常见问题

在本节我们分别回答这些问题，然后通过一个具体的实验来将这些相关的知识串联成一个整体。

3.7.1 Kubernetes网络模型

Kubernetes网络模型设计的一个基础原则是：每个Pod都拥有一个独立的IP地址，而且假定所有Pod都在一个可以直接连通的、扁平的网络空间中。所以不管它们是否运行在同一个Node（宿主机）中，都要求它们可以直接通过对方的IP进行访问。设计这个原则的原因是，用户不需要额外考虑如何建立Pod之间的连接，也不需要考虑将容器端口映射到主机端口等问题。

实际上在Kubernetes的世界里，IP是以Pod为单位进行分配的。一个Pod内部的所有容器共享一个网络堆栈（实际上就是一个网络命名空间，包括它们的IP地址、网络设备、配置等都是共享的）。按照这个网络原则抽象出来的一个Pod一个IP的设计模型也被称作IP-per-Pod模型。

由于Kubernetes的网络模型假设Pod之间访问时使用的是对方Pod的实际地址，所以一个Pod内部的应用程序看到的自己的IP地址和端口与集群内其他Pod看到的一样。它们都是Pod实际分配的IP地址（从docker0上分配的）。将IP地址和端口在Pod内部和外部都保持一致，我们可以不使用NAT来进行转换，地址空间也自然是平的。Kubernetes的网络之所以这么设计，主要原因就是可以兼容过去的应用。当然，我们使用Linux命令“ip addr show”也能看到这些地址，和程序看到的没有什么区别。所以这种IP-per-Pod的方案很好地利用了现有的各种域名解析和发现机制。

一个Pod一个IP的模型还有另外一层含义，那就是同一个Pod内的不同容器将会共享一个网络命名空间，也就是说同一个Linux网络协议栈。这就意味着同一个Pod内的容器可以通过localhost来连接对方的端口。这种关系和同一个VM内的进程之间的关系是一样的，看起来Pod内的容器之间的隔离性降低了，而且Pod内不同容器之间的端口是共享的，没有所谓的私有端口的概念了。如果你的应用必须要使用一些特定的端口范围，那么你也可以为这些应用单独创建一些Pod。反之，对那些没有特殊需要的应用，这样做的好处是Pod内的容器是共享部分资源的，通过共享资源互相通信显然更加容易和高效。针对这些应用，虽然损失了可接受范围内的部分隔离性，但也是值得的。

IP-per-Pod模式和Docker原生的通过动态端口映射方式实现的多节点访问模式有什么区别呢？主要区别是后者的动态端口映射会引入端口管理的复杂性，而且访问者看到的IP地址和端口与服务提供者实际绑定的不同（因为NAT的缘故，它们都被映射成新的地址或端口了），这也会引起应用配置的复杂化。同时，标准的DNS等名字解析服务也不适用了。甚至服务注册和发现机制都将受到挑战，因为在端口映射情况下，服务自身很难知道自己对外暴露的真实的服务IP和端口。而外部应用也无法通过服务所在容器的私有IP地址和端口来访问服务。

总的来说，IP-per-Pod模型是一个简单的兼容性较好的模型。从该模型的网络的端口分配、域名解析、服务发现、负载均衡、应用配置和迁移等角度来看，Pod都能够被看作一台独立的“虚拟机”或“物理机”。

按照这个网络抽象原则，Kubernetes对网络有什么前提和要求呢？

Kubernetes对集群的网络有如下要求:

- (1) 所有容器都可以在不用NAT的方式下同别的容器通信;
- (2) 所有节点都可以在不用NAT的方式下同所有容器通信, 反之亦然;
- (3) 容器的地址和别人看到的地址是同一个地址。

这些基本的要求意味着并不是只要两台机器运行 Docker , Kubernetes就可以工作了。具体的集群网络实现必须保障上述基本要求, 原生的Docker网络目前还不能很好地支持这些要求。

实际上, 这些对网络模型的要求并没有降低整个网络系统的复杂度。如果你的程序原来在VM上运行, 而那些VM拥有独立IP, 并且它们之间可以直接透明地通信, 那么Kubernetes的网络模型就和VM使用的网络模型是一样的。所以使用这种模型可以很容易地将已有的应用程序从VM或者物理机迁移到容器上。

当然, 谷歌设计Kubernetes的一个主要运行基础就是其云环境GCE (Google Compute Engine), 在GCE下这些网络要求都是默认支持的。另外, 常见的其他公有云服务商如亚马逊等, 在它们的公有云计算环境下也是默认支持这个模型的。

由于部署私有云的场景会更普遍, 所以在私有云中运行Kubernetes+Docker集群之前, 就需要自己搭建出符合Kubernetes要求的网络环境。现在的开源世界有很多开源组件可以帮助我们打通Docker容器和容器之间的网络, 实现Kubernetes要求的网络模型。当然

每种方案都有适合的场景，我们要根据自己的实际需要进行选择。在后面的章节中会对常见的开源方案进行介绍。

Kubernetes的网络依赖于Docker，Docker的网络又离不开Linux操作系统内核特性的支持，所以我们有必要先深入了解Docker背后的网络原理和基础知识。接下来我们一起深入学习一些必要的Linux网络知识。

3.7.2 Docker的网络基础

Docker本身的技术依赖于近年Linux内核虚拟化技术的发展，所以Docker对Linux内核的特性有很强的依赖。这里将Docker使用到的与Linux网络有关的主要技术进行简要介绍，这些技术包括如下几种，如图3.16所示。



图3.16 Docker使用到的与Linux网络有关的主要技术

1.网络的命名空间

为了支持网络协议栈的多个实例，Linux在网络栈中引入了网络命名空间（Network Namespace），这些独立的协议栈被隔离到不同的命名空间中。处于不同命名空间的网络栈是完全隔离的，彼此之间无法通信，就好像两个“平行宇宙”。通过这种对网络资源的隔离，就能在

一个宿主机上虚拟多个不同的网络环境。而Docker也正是利用了网络的命名空间特性，实现了不同容器之间网络的隔离。

在Linux的网络命名空间内可以有自己独立的路由表及独立的Iptables/Netfilter设置来提供包转发、NAT及IP包过滤等功能。

为了隔离出独立的协议栈，需要纳入命名空间的元素有进程、套接字、网络设备等。进程创建的套接字必须属于某个命名空间，套接字的操作也必须在命名空间内进行。同样，网络设备也必须属于某个命名空间。因为网络设备属于公共资源，所以可以通过修改属性实现在命名空间之间移动。当然，是否允许移动和设备的特征有关。

让我们稍微深入Linux操作系统内部，看它是如何实现网络命名空间的，这也会对理解后面的概念有帮助。

1) 网络命名空间的实现

Linux的网络协议栈是十分复杂的，为了支持独立的协议栈，相关的这些全局变量都必须修改为协议栈私有。最好的办法就是让这些全局变量成为一个Net Namespace变量的成员，然后为协议栈的函数调用加入一个Namespace参数。这就是Linux实现网络命名空间的核心。

同时，为了保证对已经开发的应用程序及内核代码的兼容性，内核代码隐式地使用了命名空间内的变量。我们的程序如果没有对命名空间的特殊需求，那么不需要写额外的代码，网络命名空间对应用程序而言是透明的。

在建立了新的网络命名空间，并将某个进程关联到这个网络命名空间后，就出现了类似于如图3.17所示的内核数据结构，所有网络栈

变量都放入了网络命名空间的数据结构中。这个网络命名空间是属于它的进程组私有的，和其他进程组不冲突。

新生成的私有命名空间只有回环lo设备（而且是停止状态），其他设备默认都不存在，如果我们需要，则要一一手工建立。**Docker**容器中的各类网络栈设备都是**Docker Daemon**在启动时自动创建和配置的。

所有的网络设备（物理的或虚拟接口、桥等在内核里都叫作**NetDevice**）都只能属于一个命名空间。当然，通常物理的设备（连接实际硬件的设备）只能关联到**root**这个命名空间中。虚拟的网络设备（虚拟的以太网接口或者虚拟网口对）则可以被创建并关联到一个给定的命名空间中，而且可以在这些命名空间之间移动。

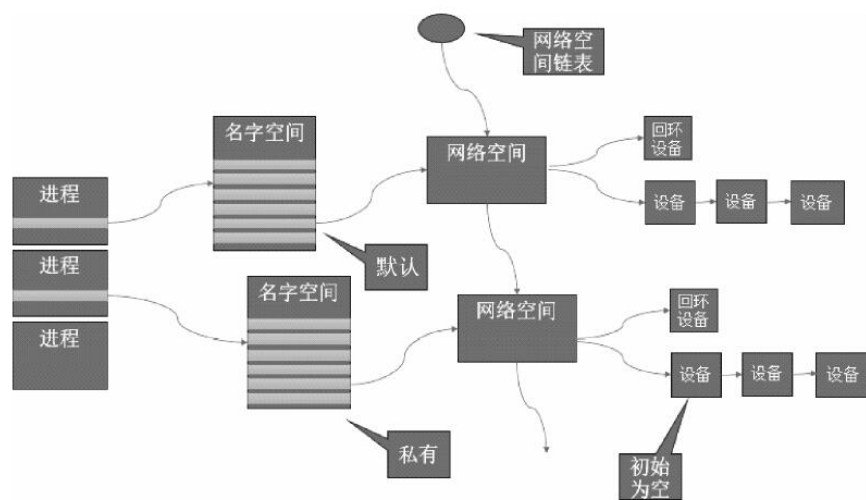


图3.17 命名空间内核结构

前面我们提到，由于网络命名空间代表的是一个独立的协议栈，所以它们之间是相互隔离的，彼此无法通信，在协议栈内部都看不到对方。那么有没有办法打破这种限制，让处于不同命名空间的网络相

互通信，甚至和外部的网络进行通信呢？答案就是“Veth 设备对”。“Veth设备对”的一个重要作用就是打通互相看不到的协议栈之间的壁垒，它就像一个管子，一端连着这个网络命名空间的协议栈，一端连着另一个网络命名空间的协议栈。所以如果想在两个命名空间之间进行通信，就必须有一个Veth设备对。后面我们会介绍如何操作Veth设备对来打通不同命名空间之间的网络。

2) 网络命名空间操作

下面列举一些网络命名空间的操作。

我们可以使用Linux iproute2系列配置工具中的IP命令来操作网络命名空间。注意，这个命令需要由root用户运行。

创建一个命名空间：

```
ip netns add <name>
```

在命名空间内执行命令：

```
ip netns exec <name><command>
```

如果想执行多个命令，则可以先进入内部的sh，然后执行：

```
ip netns exec <name> bash
```

之后就是新的命名空间内进行操作了。退出到外面的命名空间时，请输入“exit”。

3) 网络命名空间的一些技巧

操作网络命名空间时的一些实用技巧如下。

我们可以在不同的网络命名空间之间转移设备，例如下面会提到的Veth设备对的转移。因为一个设备只能属于一个命名空间，所以转移后在这个命名空间内就看不到这个设备了。具体哪些设备能够转移到不同的命名空间呢？在设备里面有一个重要的属性：NETIF_F_ETNS_LOCAL，如果这个属性为“on”，则不能转移到其他命名空间内。Veth设备属于可以转移的设备，而很多其他设备如lo设备、vxlan设备、ppp设备、bridge设备等都是不可以转移的。至于将无法转移的设备移动到别的命名空间的操作，则会得到无效参数的错误提示。

```
# ip link set br0 netns ns1
RTNETLINK answers: Invalid argument
```

如何知道这些设备是否可以转移呢？可以使用ethtool工具查看：

```
# ethtool -k br0
netns-local: on [fixed]
```

netns-local的值是on，就说明不可以转移，否则可以。

2.Veth设备对

引入Veth设备对是为了在不同的网络命名空间之间进行通信，利用它可以直接将两个网络命名空间连接起来。由于要连接两个网络命

名空间，所以Veth设备都是成对出现的，很像一对以太网卡，并且中间有一根直连的网线。既然是一对网卡，那么我们将其中一端称为另一端的peer。在Veth设备的一端发送数据时，它会将数据直接发送到另一端，并触发另一端的接收操作。

整个Veth的实现非常简单，有兴趣的读者可以参考源代码“drivers/net/veth.c”的实现。图3.18是Veth设备对的示意图。

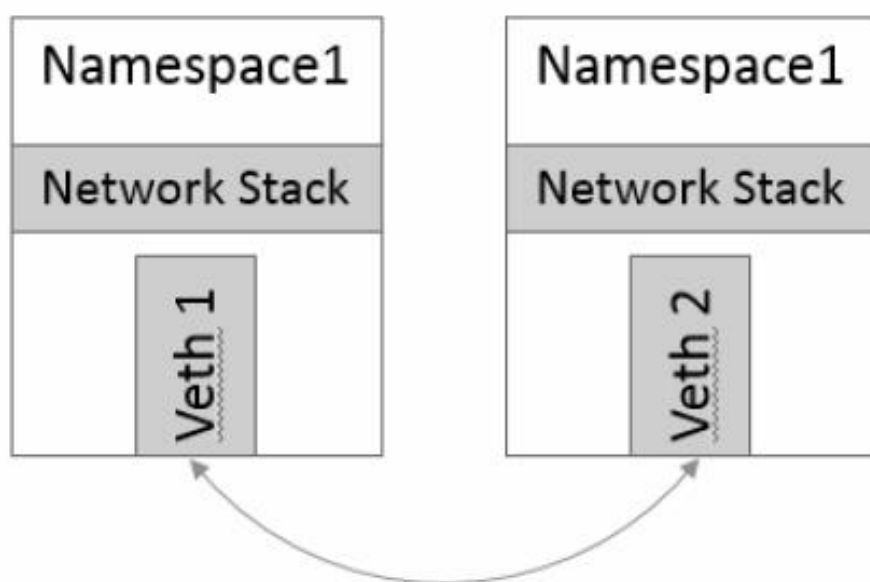


图3.18 eth设备对示意图

1) Veth设备对的操作命令

接下来看看如何创建Veth设备对，如何连接到不同的命名空间，并设置它们的地址，让它们通信。

创建Veth设备对：

```
ip link add veth0 type veth peer name veth1
```

创建后，可以查看veth设备对的信息。使用ip link show命令查看所有网络接口：

```
# ip link show

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
state UNKNOWN mode DEFAULT
    Link/loopback: 00:00:00:00:00:00 brd
00:00:00:00:00:00

2: eno16777736: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu
1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
    link/ether 00:0c:29:cf:1a:2e brd ff:ff:ff:ff:ff:ff

3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu
1500 qdisc noqueue state UP mode DEFAULT
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff

19: veth1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop
state DOWN mode DEFAULT qlen 1000
    link/ether 7e:4a:ae:41:a3:65 brd ff:ff:ff:ff:ff:ff

20: veth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop
state DOWN mode DEFAULT qlen 1000
    link/ether ea:da:85:a3:75:8a brd ff:ff:ff:ff:ff:ff
```

看到了吧，有两个设备生成了，一个是veth0，它的peer是veth1。

现在这两个设备都在自己的命名空间内，那怎么能行呢？好了，如果将Veth看作有两个头的网线，那么我们将另一个头甩给另一个命

名空间吧:

```
ip link set veth1 netns netns1
```

这时可在外面这个命名空间内看两个设备的情况:

```
# ip link show

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
state UNKNOWN mode DEFAULT
    Link/loopback:  00:00:00:00:00:00   brd
00:00:00:00:00:00

2: eno16777736: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu
1500 qdisc pfifo_fast state UP mode DEFAULT qlen 1000
    link/ether 00:0c:29:cf:1a:2e brd ff:ff:ff:ff:ff:ff

3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu
1500 qdisc noqueue state UP mode DEFAULT
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff

20: veth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop
state DOWN mode DEFAULT qlen 1000
    link/ether ea:da:85:a3:75:8a brd ff:ff:ff:ff:ff:ff
```

只剩一个veth0设备了，已经看不到另一个设备了，另一个设备已经转移到另一个网络命名空间了。

在netns1网络命名空间中可以看到veth1设备了，符合预期。

```
# ip netns exec netns1 ip link show

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
```

```
state UNKNOWN mode DEFAULT
                Link/loopback:  00:00:00:00:00:00   brd
00:00:00:00:00:00
        19: veth1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop
state DOWN mode DEFAULT qlen 1000
        link/ether 7e:4a:ae:41:a3:65 brd ff:ff:ff:ff:ff:ff
```

现在看到的结果是，两个不同的命名空间各自有一个Veth的“网线头”，各显示为一个Device（在Docker的实现里面，它除了将Veth放入容器内，还将它的名字改成了eth0，简直以假乱真，你以为它是一个本地网卡吗）。

现在可以通信了吗？不行，因为它们还没有任何地址，现在我们来给它们分配IP地址吧：

```
ip netns exec netns1 ip addr add 10.1.1.1/24 dev veth1
ip addr add 10.1.1.2/24 dev veth0
```

再启动它们：

```
ip netns exec netns1 ip link set dev veth1 up
ip link set dev veth0 up
```

现在两个网络命名空间可以互相通信了：

```
# ping 10.1.1.1
PING10.1.1.1 (10.1.1.1) 56(84) bytes of data.
 64 bytes from 10.1.1.1: icmp_seq=1 ttl=64 time=0.035
```

```

ms
    64 bytes from 10.1.1.1: icmp_seq=2 ttl=64 time=0.096
ms
    ^C
    --- 10.1.1.1 ping statistics ---
    2 packets transmitted, 2 received, 0% packet loss,
time 1001ms
    rtt min/avg/max/mdev = 0.035/0.065/0.096/0.031 ms

# ip netns exec netns1 ping 10.1.1.2
PING10.1.1.2 (10.1.1.2) 56(84) bytes of data.
    64 bytes from 10.1.1.2: icmp_seq=1 ttl=64 time=0.045
ms
    64 bytes from 10.1.1.2: icmp_seq=2 ttl=64 time=0.105
ms
    ^C
    --- 10.1.1.2 ping statistics ---
    2 packets transmitted, 2 received, 0% packet loss,
time 1000ms
    rtt min/avg/max/mdev = 0.045/0.075/0.105/0.030 ms

```

至此，两个网络命名空间之间就完全通了。

至此我们就能够理解Veth设备对的原理和用法了。在Docker内部，Veth设备对也是联系容器到外面的重要设备，离开它是不行的。

2) Veth设备对如何查看对端

我们在操作Veth设备对的时候有一些实用技巧，如下所示。

一旦将Veth设备对的peer端放入另一个命名空间，我们在本命名空间内就看不到它了。那么我们怎么知道这个Veth对的对端在哪里呢，也就是说它到底连接到哪个别的命名空间呢？可以使用ethtool工具来查看（当网络命名空间特别多的时候，这可不是一件很容易的事情）。

首先我们在一个命名空间中查询Veth设备对端接口在设备列表中的序列号：

```
ip netns exec netns1 ethtool -S veth1
NIC statistics:
    peer_ifindex: 5
```

得知另一端的接口设备的序列号是5，我们再到另一个命名空间中查看序列号5代表什么设备：

```
ip netns exec netns2 ip link | grep 5    <-- 我们只关注
序列号是5的设备
veth0
```

好了，我们现在就找到下标为5的设备了，它是veth0，它的另一端自然就是另一个命名空间中的veth1了，因为它们互为peer。

3.网桥

Linux可以支持很多不同的端口，这些端口之间当然应该能够通信，如何将这些端口连接起来并实现类似交换机那样的多对多通信呢？这就是网桥的作用了。网桥是一个二层网络设备，可以解析收发的报文，读取目标**MAC**地址的信息，和自己记录的**MAC**表结合，来决策报文的转发端口。为了实现这些功能，网桥会学习源**MAC**地址（二层网桥转发的依据就是**MAC**地址）。在转发报文的时候，网桥只需要向特定的网络接口进行转发，从而避免不必要的网络交互。如果它遇到一个自己从未学习到的地址，就无法知道这个报文应该从哪个网口设备转发，于是只好将报文广播给所有的网络设备端口（报文来源的那个端口除外）。

在实际网络中，网络拓扑不可能永久不变。如果设备移动到另一个端口上，而它没有发送任何数据，那么网桥设备就无法感知到这个变化，结果网桥还是向原来的端口转发数据包，在这种情况下数据就会丢失。所以网桥还要对学习到的**MAC**地址表加上超时时间（默认为5分钟）。如果网桥收到了对应端口**MAC**地址回发的包，则重置超时时间，否则过了超时时间后，就认为那个设备已经不在那个端口上了，它就会重新广播发送。

在**Linux**的内部网络栈里面实现的网桥设备，作用和上面的描述相同。过去**Linux**主机一般都只有一个网卡，现在多网卡的机器越来越多，而且还有很多虚拟的设备存在，所以**Linux**的网桥提供了这些设备之间互相转发数据的二层设备。

Linux内核支持网口的桥接（目前只支持以太网接口）。但是与单纯的交换机不同，交换机只是一个二层设备，对于接收到的报文，要么转发，要么丢弃。运行着**Linux**内核的机器本身就是一台主机，有可能是网络报文的目的地，其收到的报文除了转发和丢弃，还可能被送

到网络协议栈的上层（网络层），从而被自己（这台主机本身的协议栈）消化，所以我们既可以把网桥看作一个二层设备，也可以看作一个三层设备。

1) Linux网桥的实现

Linux内核是通过一个虚拟的网桥设备（Net Device）来实现桥接的。这个虚拟设备可以绑定若干个以太网接口设备，从而将它们桥接起来。如图3.19所示，这种Net Device网桥和普通的设备不同，最明显的一个特性是它还可以有一个IP地址。

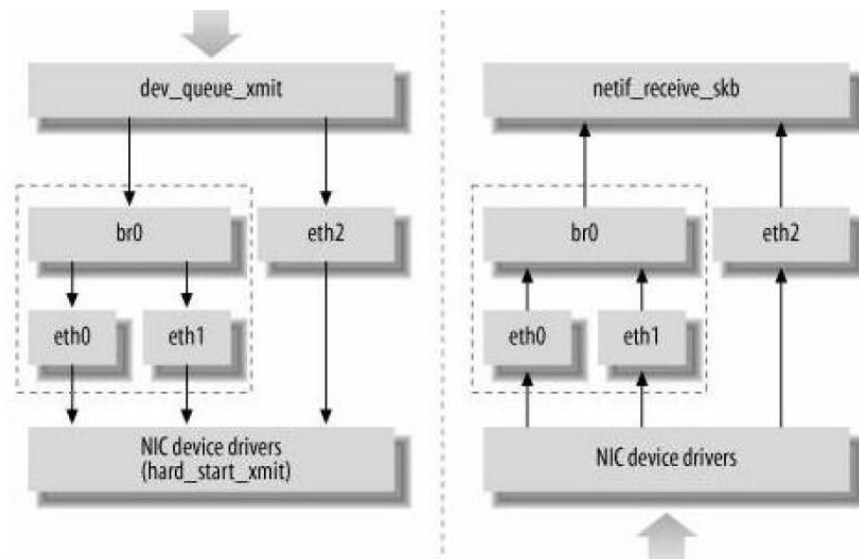


图3.19 网桥的位置

如图3.19所示，网桥设备br0绑定了eth0和eth1。对于网络协议栈的上层来说，只看得到br0。因为桥接是在数据链路层实现的，上层不需要关心桥接的细节，于是协议栈上层需要发送的报文被送到br0，网桥设备的处理代码判断报文该被转发到eth0还是eth1，或者两者皆转发；

反过来，从eth0或从eth1接收到的报文被提交给网桥的处理代码，在这里会判断报文应该被转发、丢弃还是提交到协议栈上层。

而有时eth0、eth1也可能会作为报文的源地址或目的地址，直接参与报文的发送与接收，从而绕过网桥。

2) 网桥的常用操作命令

Docker自动完成了对网桥的创建和维护。为了进一步理解网桥，下面举几个常用的网桥操作例子，对网桥进行手工操作：

```
#brctl addbr xxxxx
```

就是新增一个网桥

之后可以增加端口，在Linux中，一个端口其实就是一个物理网卡。将物理网卡和网桥连接起来：

```
#brctl addif xxxxx ethx
```

网桥的物理网卡作为一个端口，由于在链路层工作，就不再需要IP地址了，这样上面的IP地址自然失效：

```
#ifconfig ethx 0.0.0.0
```

给网桥配置一个IP地址：

```
#ifconfig brxxx xxx.xxx.xxx.xxx
```

这样网桥就有了一个IP地址，而连接到上面的网卡就是一个纯链路层设备了。

4.Iptables/Netfilter

我们知道，Linux网络协议栈非常高效，同时比较复杂。如果我们希望在数据的处理过程中对关心的数据进行一些操作该怎么做呢？Linux提供了一套机制来为用户实现自定义的数据包处理过程。

在Linux网络协议栈中有一组回调函数挂接点，通过这些挂接点挂接的钩子函数可以在Linux网络栈处理数据包的过程中对数据包进行一些操作，例如过滤、修改、丢弃等。整个挂接点技术叫作Netfilter和Iptables。

Netfilter负责在内核中执行各种挂接的规则，运行在内核模式中；而Iptables是在用户模式下运行的进程，负责协助维护内核中Netfilter的各种规则表。通过二者的配合来实现整个Linux网络协议栈中灵活的数据包处理机制。

Netfilter可以挂接的规则点有5个，如图3.20中的深色椭圆所示。

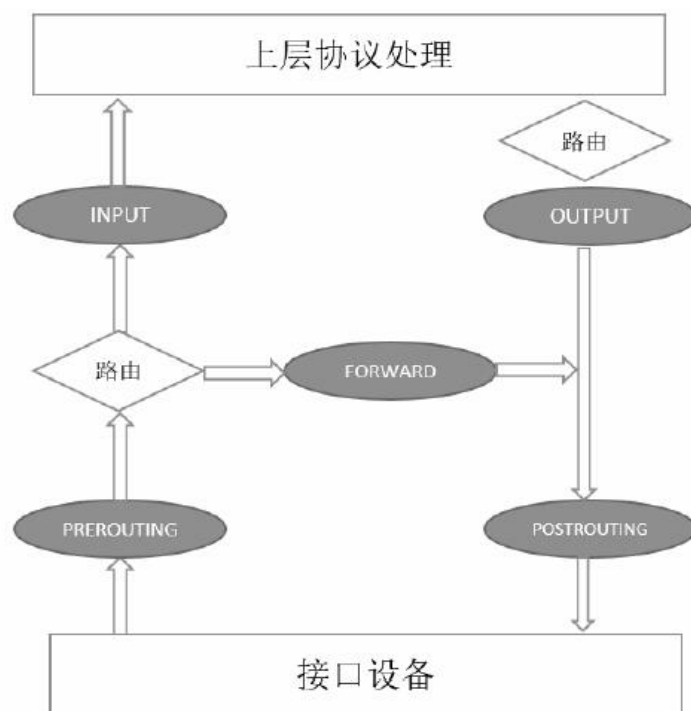


图3.20 Netfilter挂接点

1) 规则表Table

这些挂接点能挂接的规则也分不同的类型（也就是规则表Table），我们可以在不同类型的Table中加入我们的规则。目前主要支持的Table类型为：

- RAW;
- MANGLE;
- NAT;
- FILTER。

上述4个Table（规则链）的优先级是RAW最高，FILTER最低。

在实际应用中，不同的挂接点需要的规则类型通常不同。例如，在Input的挂接点上明显不需要FILTER过滤规则，因为根据目标地址，已经选择好本机的上层协议栈了，所以无须再挂接FILTER过滤规则。目前Linux系统支持的不同挂接点能挂接的规则类型如图3.21所示。

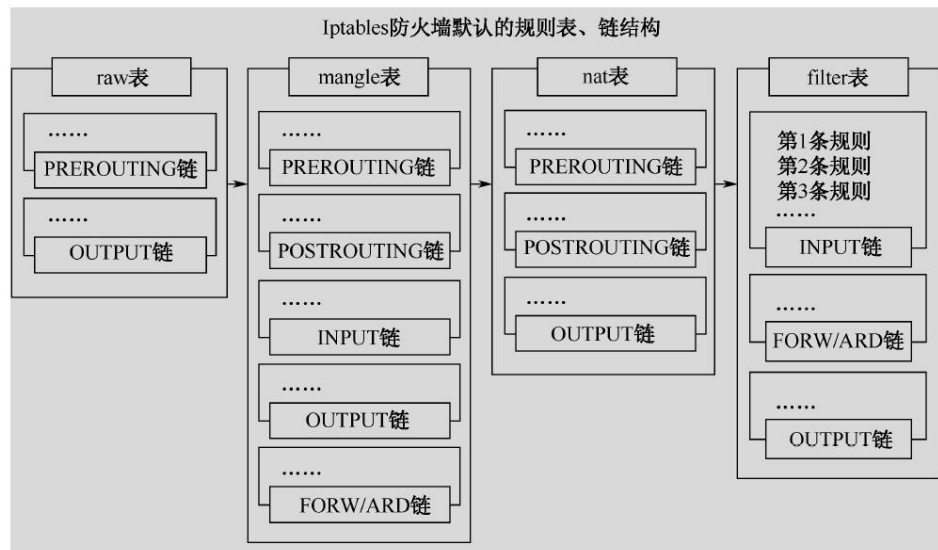


图3.21 不同表的挂接点

当Linux协议栈的数据处理运行到挂接点时，它会依次调用挂接点上所有的挂钩函数，直到数据包的处理结果是明确地接受或者拒绝。

2) 处理规则

每个规则的特性都分为以下几部分：

- 表类型（准备干什么事情）；
- 什么挂接点（什么时候起作用）；
- 匹配的参数是什么（针对什么样的数据包）；
- 匹配后有什么动作（匹配后具体的操作是什么）。

表类型和什么挂接点在前面已经介绍了，现在我们看看匹配的参数和匹配后的动作。

（1）匹配的参数

匹配的参数用于对数据包或者TCP数据连接的状态进行匹配。当有多个条件存在时，它们一起起作用，来达到只针对某部分数据进行修改的目的。常见的匹配参数有：

- 流入、流出的网络接口；
- 来源、目的地址；
- 协议类型；
- 来源、目的端口。

（2）匹配后的动作

一旦有数据匹配上，就会执行相应的动作。动作类型既可以是标准的预定义的几个动作，也可以是自定义的模块注册的动作，或者是一个新的规则链，以便更好地组织一组动作。

3) Iptables命令

Iptables 命令用于协助用户维护各种规则。我们在使用Kubernetes、Docker的过程中，通常都会去查看相关的Netfilter配置。这里只介绍如何查看规则表，详细的介绍请参照Linux的Iptables帮助文档。

查看系统中已有的规则的方法如下。

- iptables-save: 按照命令的方式打印Iptables的内容。

- **Iptables-vnL**: 以另一种格式显示**Netfilter**表的内容。

5.路由

Linux系统包含一个完整的路由功能。当**IP**层在处理数据发送或者转发的时候，会使用路由表来决定发往哪里。通常情况下，如果主机与目的主机直接相连，那么主机可以直接发送**IP**报文到目的主机，这个过程比较简单。例如，通过点对点的链接或通过网络共享，如果主机与目的主机没有直接相连，那么主机会将**IP**报文发送给默认的路由器，然后由路由器来决定往哪发送**IP**报文。

路由功能由**IP**层维护的一张路由表来实现。当主机收到数据报文时，它用此表来决策接下来应该做什么操作。当从网络侧接收到数据报文时，**IP**层首先会检查报文的**IP**地址是否与主机自身的地址相同。如果数据报文中的**IP**地址是主机自身的地址，那么报文将被发送到传输层相应的协议中去。如果报文中的**IP**地址不是主机自身的地址，并且主机配置了路由功能，那么报文将被转发，否则，报文将被丢弃。

路由表中的数据一般是以条目形式存在的。一个典型的路由表条目通常包含以下主要的条目项。

(1) 目的**IP**地址: 此字段表示目标的**IP**地址。这个**IP**地址可以是某台主机的地址，也可以是一个网络地址。如果这个条目包含的是一个主机地址，那么它的主机**ID**将被标记为非零；如果这个条目包含的是一个网络地址，那么它的主机**ID**将被标记为零。

(2) 下一个路由器的**IP**地址: 为什么采用“下一个”的说法，是因为下一个路由器并不总是最终的目的路由器，它很可能是一个中间路

由器。条目给出下一个路由器的地址用来转发从相应接口接收到的IP数据报文。

(3) 标志：这个字段提供了另一组重要信息，例如目的IP地址是一个主机地址还是一个网络地址。此外，从标志中可以得知下一个路由器是一个真实路由器还是一个直接相连的接口。

(4) 网络接口规范：为一些数据报文的网络接口规范，该规范将与报文一起被转发。

在通过路由表转发时，如果任何条目的第1个字段完全匹配目的IP地址（主机）或部分匹配条目的IP地址（网络），那么它将指示下一个路由器的IP地址。这是一个重要的信息，因为这些信息直接告诉主机（具备路由功能的）数据包应该转发到哪个“下一个路由器”去。而条目中的所有其他字段将提供更多的辅助信息来为路由转发做决定。

如果没有找到一个完全匹配的IP，那么就接着搜索相匹配的网络ID。如果找到，那么该数据报文会被转发到指定的路由器上。可以看出，网络上的所有主机都通过这个路由表中的单个（这个）条目进行管理。

如果上述两个条件都不匹配，那么该数据报文将被转发到一个默认路由器上。

如果上述步骤失败，默认路由器也不存在，那么该数据报文最终无法被转发。任何无法投递的数据报文都将产生一个ICMP主机不可达或ICMP网络不可达的错误，并将此错误返回给生成此数据报文的应用程序。

1) 路由表的创建

Linux的路由表至少包括两个表（当启用策略路由的时候，还会有其他表）：一个是LOCAL，另一个是MAIN。在LOCAL表中会包含所有的本地设备地址。LOCAL路由表是在配置网络设备地址时自动创建的。LOCAL表用于供Linux协议栈识别本地地址，以及进行本地各个不同网络接口之间的数据转发。

可以通过下面的命令查看LOCAL表的内容：

```
# ip route show table local type local
10.1.1.0 dev flannel0 proto kernel scope host src
10.1.1.0
127.0.0.0/8 dev lo proto kernel scope host src
127.0.0.1
127.0.0.1 dev lo proto kernel scope host src
127.0.0.1
172.17.42.1 dev docker proto kernel scope host src
172.17.42.1
192.168.1.128 dev eno16777736 proto kernel scope
host src 192.168.1.128
```

MAIN表用于各类网络IP地址的转发。它的建立既可以使用静态配置生成，也可以使用动态路由发现协议生成。动态路由发现协议一般使用组播功能来通过发送路由发现数据，动态地交换和获取网络的路由信息，并更新到路由表中。

Linux下支持路由发现协议的开源软件有许多，常用的有Quagga、Zebra等。第4章会介绍使用Quagga动态容器路由发现的机制来实现Kubernetes的网络组网。

2) 路由表的查看

我们可以使用ip route list命令查看当前的路由表。

```
# ip route list
192.168.6.0/24 dev eno16777736 proto kernel scope
link src 192.168.6.140 metric 1
```

在上面的例子代码中，只有一个子网的路由，源地址是192.168.6.140（本机），目标地址是192.168.6.0/24网段的数据，都将通过eth0接口设备发送出去。

Netstat-rn是另一个查看路由表的工具：

```
# netstat -rn
Kernel IP routing table
      Destination          Gateway          Genmask
Flags  MSS Window      irtt Iface
      0.0.0.0              192.168.6.2      0.0.0.0          UG
0 0
      192.168.6.0          0.0.0.0          255.255.255.0    U
0 0
      0 eth0
      0 eth0
```

在它显示的信息中，如果标志是**U**，则说明是可达路由；如果标志是**G**，则说明这个网络接口连接的是网关，否则说明是直连主机。

3.7.3 Docker的网络实现

标准的Docker支持以下4类网络模式。

- host模式：使用--net=host指定。
- container模式：使用--net=container: NAME_or_ID指定。
- none模式：使用--net=none指定。
- bridge模式：使用--net=bridge指定，为默认设置。

在Kubernetes管理模式下，通常只会使用bridge模式，所以本节只介绍bridge模式下Docker是如何支持网络的。

在bridge模式下，Docker Daemon第1次启动时会创建一个虚拟的网桥，默认的名字是docker0，然后按照RPC1918的模型，在私有网络空间中给这个网桥分配一个子网。针对由Docker创建出来的每一个容器，都会创建一个虚拟的以太网设备（Veth设备对），其中一端关联到网桥上，另一端使用Linux的网络命名空间技术，映射到容器内的eth0设备，然后从网桥的地址段内给eth0接口分配一个IP地址。

如图3.22所示就是Docker的默认桥接网络模型。

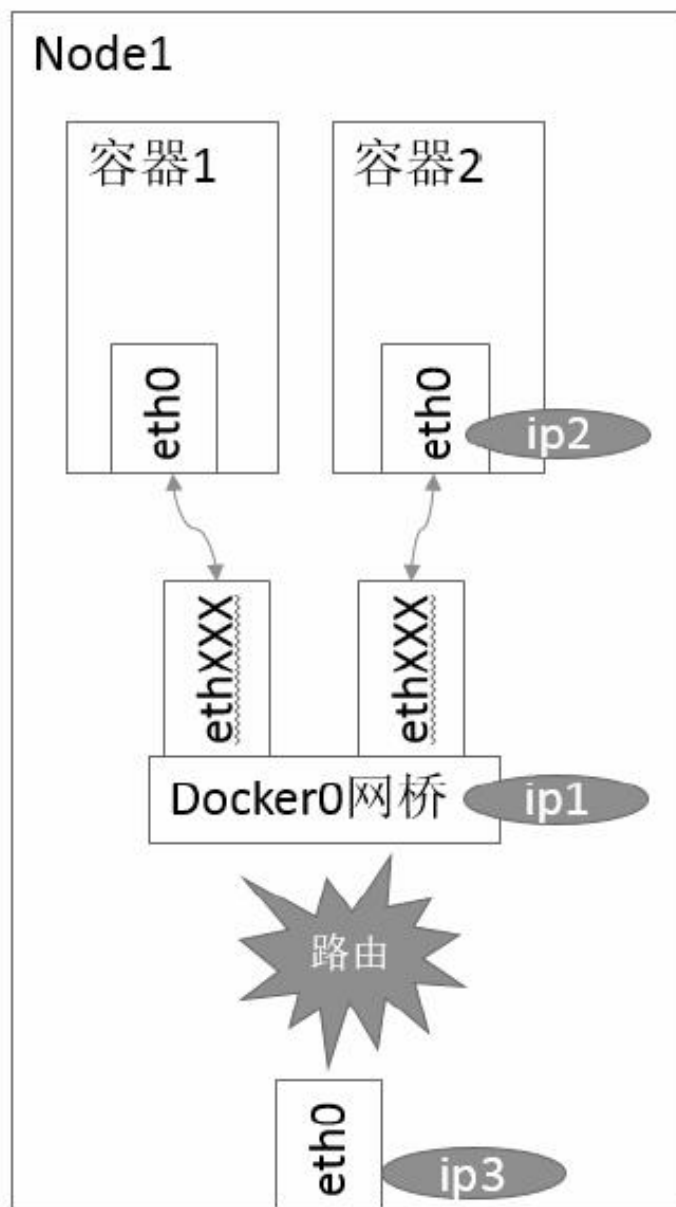


图3.22 默认的Docker网络桥接模型

其中ip1是网桥的IP地址，Docker Daemon会在几个备选地址段里给它选一个，通常是172开头的地址。这个地址和主机的IP地址是不重叠的。ip2是Docker在启动容器的时候，在这个地址段随机选择的一个没有使用的IP地址，Docker占用它并分配给了被启动的容器。相

应的MAC地址也根据这个IP地址，在02: 42: ac: 11: 00: 00和02: 42: ac: 11: ff: ff的范围内生成，这样做可以确保不会有ARP的冲突。

启动后，Docker还将Veth对的名字映射到了eth0网络接口。ip3就是主机的网卡地址。

在一般情况下，ip1、ip2和ip3是不同的IP段，所以在默认不做任何特殊配置的情况下，在外部是看不到ip1和ip2的。

这样做的结果就是，同一台机器内的容器之间可以相互通信。不同主机上的容器不能够相互通信。实际上它们甚至有可能在相同的网络地址范围内（不同的主机上的docker0的地址段可能是一样的）。

为了让它们跨节点互相通信，就必须在主机的地址上分配端口，然后通过这个端口路由或代理到容器上。这种做法显然意味着一定要在容器之间小心谨慎地协调好端口的分配，或者使用动态端口的分配技术。在不同应用之间协调好端口分配是十分困难的事情，特别是集群水平扩展的时候。而动态的端口分配也会带来高度复杂性，例如：每个应用程序都只能将端口看作一个符号（因为是动态分配的，无法提前设置）。而且API Server也要在分配完后，将动态端口插入到配置的合适位置。另外，服务也必须能互相之间找到对方等。这些都是Docker的网络模型在跨主机访问时面临的问题。

1) 查看Docker启动后的系统情况

我们已经知道，Docker网络在bridge模式下Docker Daemon启动时创建docker0网桥，并在网桥使用的网段为容器分配IP。让我们看看实际的操作。

在刚刚启动DockerDaemon并且还没有启动任何容器的时候，网络协议栈的配置情况如下：

```
# systemctl start docker

# ip addr

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd
00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever

2: eno16777736: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu
1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:14:3d:80 brd ff:ff:ff:ff:ff:ff
        inet 192.168.1.133/24 brd 192.168.1.255 scope
global eno16777736
            valid_lft forever preferred_lft forever
        inet6 fe80::20c:29ff:fe14:3d80/64 scope link
            valid_lft forever preferred_lft forever

3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu
1500 qdisc noqueue state DOWN
    link/ether 02:42:6e:af:0e:c3 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/24 scope global docker0
        valid_lft forever preferred_lft forever
```

```

# iptables-save

# Generated by iptables-save v1.4.21 on Thu Sep 24
17:11:04 2015

*nat
:PREROUTING ACCEPT [7:878]
:INPUT ACCEPT [7:878]
:OUTPUT ACCEPT [3:536]
:POSTROUTING ACCEPT [3:536]
:DOCKER - [0:0]
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type
LOCAL -j DOCKER
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j
MASQUERADE
COMMIT

# Completed on Thu Sep 24 17:11:04 2015

# Generated by iptables-save v1.4.21 on Thu Sep 24
17:11:04 2015

*filter
:INPUT ACCEPT [133:11362]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [37:5000]
:DOCKER - [0:0]
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -o docker0 -m conntrack --ctstate
RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT

```

```
-A FORWARD -i docker0 -o docker0 -j ACCEPT  
COMMIT  
# Completed on Thu Sep 24 17:11:04 2015
```

可以看到，Docker创建了docker0网桥，并添加了Iptables规则。docker0网桥和Iptables规则都处于root命名空间中。通过解读这些规则，我们发现，在还没有启动任何容器时，如果启动了Docker Daemon，那么它就已经做好了通信的准备。对这些规则的说明如下。

(1) 在NAT表中有3条记录，前两条匹配生效后，都会继续执行DOCKER链，而此时DOCKER链为空，所以前两条只是做了个框架，并没有实际效果。

(2) NAT表第3条的含义是，若本地发出的数据包不是发往docker0的，即是发往主机之外的设备的，都需要进行动态地址修改（MASQUERADE），将源地址从容器的地址（172段）修改为宿主机网卡的IP地址，之后就可以发送给外面的网络了。

(3) 在FILTER表中，第1条也是一个框架，因为后继的DOCKER链是空的。

(4) 在FILTER表中，第3条是说，docker0发出的包，如果需要Forward到非docker0的本地IP地址的设备，则是允许的，这样，docker0设备的包就可以根据路由规则中转到宿主机的网卡设备，从而访问外面的网络。

(5) FILTER表中，第4条是说，docker0的包还可以中转给docker0本身，即连接在docker0网桥上的不同容器之间的通信也是允许的。

(6) **FILTER**表中，第2条是说，如果接收到的数据包属于以前已经建立好的连接，那么允许直接通过。这样接收到的数据包自然又走回docker0，并中转到相应的容器。

除了这些Netfilter的设置，Linux的ip_forward功能也被Docker Daemon打开了：

```
# cat /proc/sys/net/ipv4/ip_forward
1
```

另外，我们还可以看到刚刚启动Docker后的Route表，和启动前没有什么不同：

```
# ip route
default via 192.168.1.2 dev eno16777736 proto static
metric 100
    172.17.0.0/16 dev docker proto kernel scope link
src 172.17.42.1
    192.168.1.0/24 dev eno16777736 proto kernel scope
link src 192.168.1.132
    192.168.1.0/24 dev eno16777736 proto kernel scope
link src 192.168.1.132 metric 100
```

2) 查看容器启动后的情况（容器无端口映射）

刚才我们看了Docker服务启动后的网络情况。现在，我们启动一个Registry容器后（不使用任何端口镜像参数），看一下网络堆栈部分相关的变化：

```
docker run --name register -d registry

# ip addr

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
state UNKNOWN

    link/loopback 00:00:00:00:00:00 brd
00:00:00:00:00:00

    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever

2: eno16777736: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu
1500 qdisc pfifo_fast state UP qlen 1000

    link/ether 00:0c:29:c8:12:5f brd ff:ff:ff:ff:ff:ff
        inet 192.168.1.132/24 brd 192.168.1.255 scope
global eno16777736
            valid_lft forever preferred_lft forever
        inet6 fe80::20c:29ff:fec8:125f/64 scope link
            valid_lft forever preferred_lft forever

3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu
1500 qdisc noqueue state DOWN

    link/ether 02:42:72:79:b8:88 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/24 scope global docker0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:7aff:fe79:b888/64 scope link
        valid_lft forever preferred_lft forever

13: veth2dc8bbd: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu
```



```

1500 qdisc noqueue master docker0 state UP
        link/ether be:d9:19:42:46:18 brd ff:ff:ff:ff:ff:ff
        inet6 fe80::bcd9:19ff:fe42:4618/64 scope link
                valid_lft forever preferred_lft forever

# iptables-save
# Generated by iptables-save v1.4.21 on Thu Sep 24
18:21:04 2015
*nat
:PREROUTING ACCEPT [14:1730]
:INPUT ACCEPT [14:1730]
:OUTPUT ACCEPT [59:4918]
:POSTROUTING ACCEPT [59:4918]
:DOCKER - [0:0]
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type
LOCAL -j DOCKER
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j
MASQUERADE
COMMIT
# Completed on Thu Sep 24 18:21:04 2015
# Generated by iptables-save v1.4.21 on Thu Sep 24
18:21:04 2015
*filter
:INPUT ACCEPT [2383:211572]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [2004:242872]

```

```

:DOCKER - [0:0]
-A FORWARD -o docker0 -j DOCKER
        -A FORWARD -o docker0 -m conntrack --ctstate
RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
COMMIT
# Completed on Thu Sep 24 18:21:04 2015

# ip route
default via 192.168.1.2 dev eno16777736 proto static
metric 100
        172.17.0.0/16 dev docker proto kernel scope link
src 172.17.42.1
        192.168.1.0/24 dev eno16777736 proto kernel scope
link src 192.168.1.132
        192.168.1.0/24 dev eno16777736 proto kernel scope
link src 192.168.1.132 metric 100

```

可以看到如下情况。

(1) 宿主机上的**Netfilter**和路由表都没有变化，说明在不进行端口映射时，**Docker**的默认网络是没有特殊处理的。相关的**NAT**和**FILTER**两个**Netfilter**链还是空的。

(2) 宿主机上的**Veth**对已经建立，并连接到了容器内。

我们再次进入刚刚启动的容器内，看看网络栈是什么情况。容器内部的IP地址和路由如下：

```
# docker exec -ti 24981a750a1a bash
[root@24981a750a1a /]# ip route
default via 172.17.42.1 dev eth0
172.17.0.0/16 dev eth0 proto kernel scope link src
172.17.0.10

[root@24981a750a1a /]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd
00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
22: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
qdisc noqueue state UP
    link/ether 02:42:ac:11:00:0a brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.10/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:a/64 scope link
        valid_lft forever preferred_lft forever
```

我们可以看到，默认停止的回环设备lo已经被启动，外面宿主机连接进来的Veth设备也被命名成了eth0，并且已经配置了地址

172.17.0.10。

路由信息表包含一条到docker0的子网路由和一条到docker0的默认路由。

3) 查看容器启动后的情况（容器有端口映射）

下面，我们用带端口映射的命令启动registry:

```
docker run --name register -d -p 1180:5000 registry
```

在启动后查看Iptables的变化。

```
# iptables-save
# Generated by iptables-save v1.4.21 on Thu Sep 24
18:45:13 2015
*nat
:PREROUTING ACCEPT [2:236]
:INPUT ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]
:DOCKER - [0:0]
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type
LOCAL -j DOCKER
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j
MASQUERADE
-A POSTROUTING -s 172.17.0.19/32 -d 172.17.0.19/32 -p
```

```

tcp -m tcp --dport 5000 -j MASQUERADE
    -A DOCKER ! -i docker0 -p tcp -m tcp --dport 1180 -j
DNAT --to-destination 172.17.0.19:5000
COMMIT
# Completed on Thu Sep 24 18:45:13 2015
# Generated by iptables-save v1.4.21 on Thu Sep 24
18:45:13 2015
*filter
:INPUT ACCEPT [54:4464]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [41:5576]
:DOCKER - [0:0]
-A FORWARD -o docker0 -j DOCKER
    -A FORWARD -o docker0 -m conntrack --ctstate
RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A DOCKER -d 172.17.0.19/32 ! -i docker0 -o docker0 -p
tcp -m tcp --dport 5000 -j ACCEPT
COMMIT
# Completed on Thu Sep 24 18:45:13 2015

```

从新增的规则可以看出，**Docker**服务在**NAT**和**FILTER**两个表内添加的两个**DOCKER**子链都是给端口映射用的。例如本例中我们需要把外面宿主机的**1180**端口映射到容器的**5000**端口。通过前面的分析我们知道，无论是宿主机接收到的还是宿主机本地协议栈发出的，目标地址是本地**IP**地址的包都会经过**NAT**表中的**DOCKER**子链。**Docker**为每

一个端口映射都在这个链上增加了到实际容器目标地址和目标端口的转换。

经过这个DNAT的规则修改后的IP包，会重新经过路由模块的判断进行转发。由于目标地址和端口已经是容器的地址和端口，所以数据自然就送到了docker0上，从而送到对应的容器内部。

当然在Forward时，也需要在Docker子链中添加一条规则，如果目标端口和地址是指定容器的数据，则允许通过。

在Docker按照端口映射的方式启动容器时，主要的不同就是上述Iptables部分。而容器内部的路由和网络设备，都和不做端口映射时一样，没有任何变化。

4) Docker的网络局限

我们从Docker对Linux网络协议栈的操作可以看到，Docker一开始没有考虑到多主机互联的网络解决方案。

Docker一直以来的理念都是“简单为美”，几乎所有尝试Docker的人，都被它“用法简单，功能强大”的特性所吸引，这也是Docker迅速走红的一个原因。

我们都知道，虚拟化技术中最为复杂的部分就是虚拟化网络技术，即使是单纯的物理网络部分，也是一个门槛很高的技能领域，通常只被少数网络工程师所掌握，所以我们可以理解，结合了物理网络的虚拟网络技术会有多难了。在Docker之前，所有接触过OpenStack的人的心里都有一个难以释怀的阴影，那就是它的网络问题，于是，Docker明智地避开这个“雷区”，让其他专业人员去用现有的虚拟化网

络技术解决Docker主机的互联问题，以免让用户觉得Docker太难了，从而放弃学习和使用Docker。

Docker成名以后，重新开始重视网络解决方案，收购了一家Docker网络解决方案公司——Socketplane，原因在于这家公司的产品广受好评，但有趣的是Socketplane的方案就是以Open vSwitch为核心的，其还为Open vSwitch提供了Docker镜像，以方便部署程序。之后，Docker开启了一个“宏伟”的虚拟化网络解决方案——Libnetwork，如图3.23所示是其概念图。

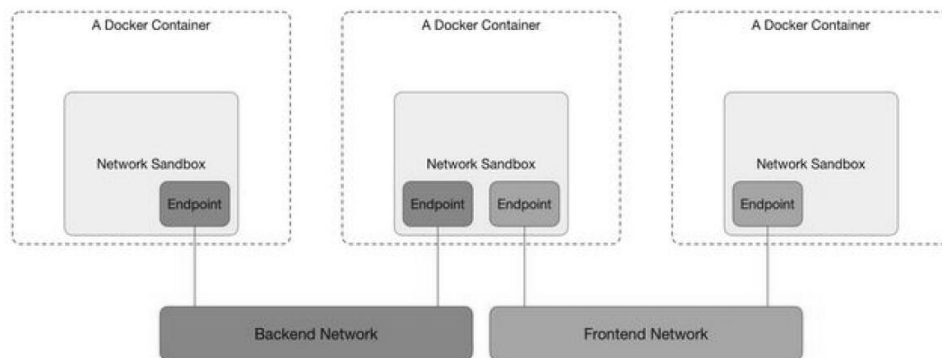


图3.23 Libnetwork概念图

这个概念图没有了IP，也没有了路由，已经颠覆了我们的网络常识了，对于不怎么懂网络的大多数人来说，它的确很有诱惑力，未来是否会对虚拟化网络的模型产生深远冲击我们还不得而知，但当前，它仅仅是Docker官方的一次“尝试”。

针对目前Docker的网络实现，Docker使用的Libnetwork组件只是将Docker平台中的网络子系统模块化为一个独立库的简单尝试，离成熟和完善还有一段距离。

所以，直到现在，仍然没有来自**Docker**官方的可以用于生产实践中的多主机网络解决方案。

3.7.4 Kubernetes的网络实现

在实际的业务场景中，业务组件之间的关系十分复杂，特别是微服务概念的推进，应用部署的粒度更加细小和灵活。为了支持业务应用组件的通信联系，Kubernetes网络的设计主要致力于解决以下场景。

- (1) 容器到容器之间的直接通信。
- (2) 抽象的Pod到Pod之间的通信。
- (3) Pod到Service之间的通信。
- (4) 集群外部与内部组件之间的通信。

其中第3条、第4条我们在之前的章节里都讲述过，本节中我们对更为基础的第1条与第2条进行深入分析和讲解。

1.容器到容器的通信

在同一个Pod内的容器（Pod内的容器是不会跨宿主机的）共享同一个网络命名空间，共享同一个Linux协议栈。所以对于网络的各类操作，就和它们在同一台机器上一样，它们甚至可以用localhost地址访问彼此的端口。

这么做的结果是简单、安全和高效，也能减少将已经存在的程序从物理机或者虚拟机移植到容器下运行的难度。在容器技术出来之前，其实大家早就积累了如何在一台机器上运行一组应用程序的经验，例如，如何让端口不冲突，以及如何让客户端发现它们等。

我们来看一下Kubernetes是如何利用Docker的网络模型的。

图3.24中的阴影部分就是在Node上运行着的一个Pod实例。在我们的例子中，容器就是图3.24中的容器1和容器2。容器1和容器2共享了一个网络的命名空间，共享一个命名空间的结果就是它们好像在一台机器上运行似的，它们打开的端口不会有冲突，可以直接使用Linux的本地IPC进行通信（例如消息队列或者管道）。其实这和传统的一组普通程序运行的环境是完全一样的，传统的程序不需要针对网络做特别的修改就可以移植了。它们之间的互相访问只需要使用localhost就可以。例如，如果容器2运行的是MySQL，那么容器1使用localhost:3306就能直接访问这个运行在容器2上的MySQL了。

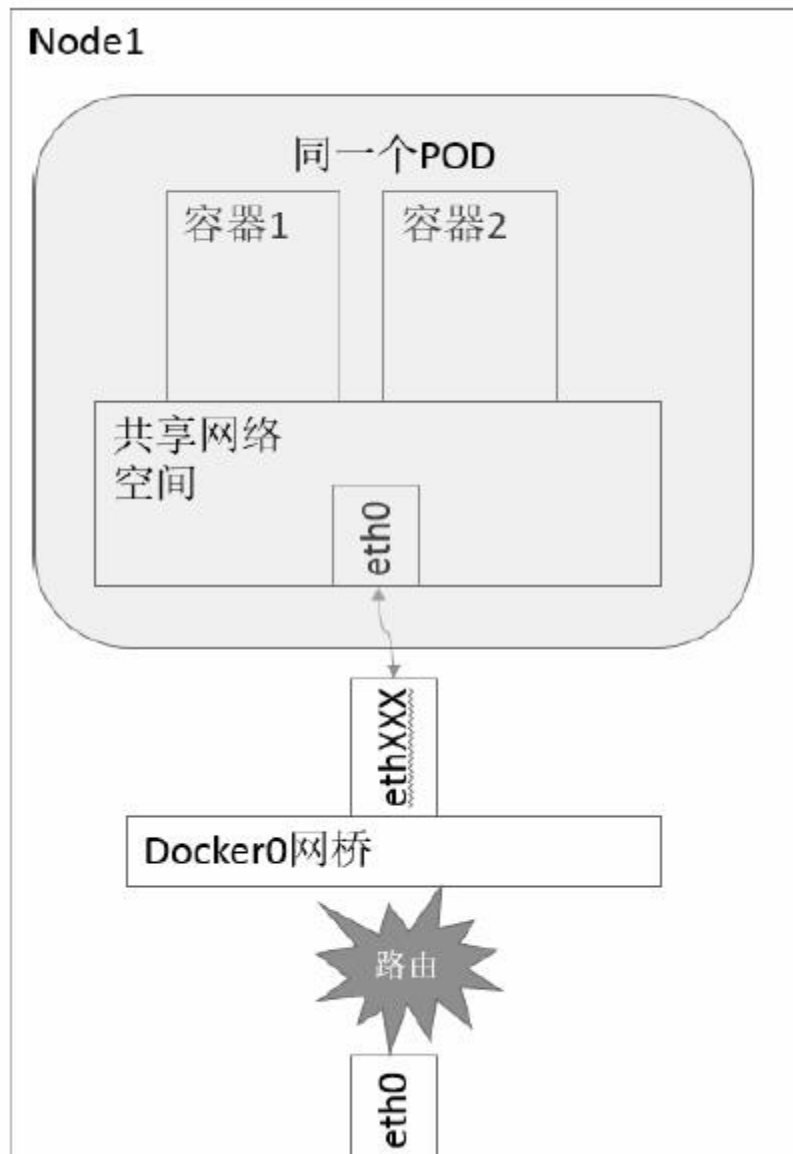


图3.24 Kubernetes的Pod网络模型

2.Pod之间的通信

我们看了同一个Pod内的容器之间的通信情况，再看看Pod之间的通信情况。

每一个Pod都有一个真实的全局IP地址，同一个Node内的不同Pod之间可以直接采用对方Pod的IP地址通信，而且不需要使用其他发现机制，例如DNS、Consul或者etcd。

Pod容器既有可能在同一个Node上运行，也有可能在不同的Node上运行，所以通信也分为两类：同一个Node内的Pod之间的通信和不同Node上的Pod之间的通信。

1) 同一个Node内的Pod之间的通信

我们看一下同一个Node上的两个Pod之间的关系，如图3.25所示。

可以看出，Pod1和Pod2都是通过Veth连接在同一个docker0网桥上的，它们的IP地址IP1、IP2都是从docker0的网段上动态获取的，它们和网桥本身的IP3是同一个网段的。

另外，在Pod1、Pod2的Linux协议栈上，默认路由都是docker0的地址，也就是说所有非本地地址的网络数据，都会被默认发送到docker0网桥上，由docker0网桥直接中转。

综上所述，由于它们都关联在同一个docker0网桥上，地址段相同，所以它们之间是能直接通信的。

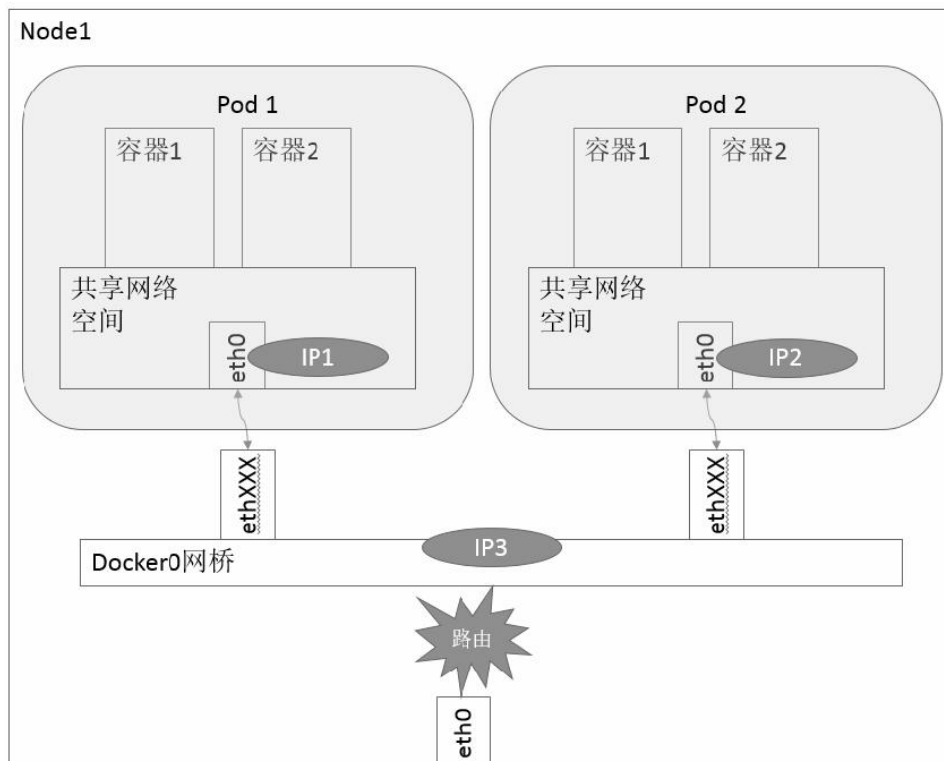


图3.25 同一个Node内的Pod关系

2) 不同Node上的Pod之间的通信

Pod的地址是与docker0在同一个网段内的，我们知道docker0网段与宿主机网卡是两个完全不同的IP网段，并且不同Node之间的通信只能通过宿主机的物理网卡进行，因此要想实现位于不同Node上的Pod容器之间的通信，就必须想办法通过主机的这个IP地址来进行寻址和通信。

另一方面，这些动态分配且藏在docker0之后的所谓“私有”IP地址也是可以找到的。Kubernetes会记录所有正在运行Pod的IP分配信息，并将这些信息保存在etcd中（作为Service的Endpoint）。这些私有IP信

息对于Pod到Pod的通信也是十分重要的，因为我们的网络模型要求Pod到Pod使用私有IP进行通信。所以首先要知道这些IP是什么。

之前提到，Kubernetes的网络对Pod的地址是平面的和直达的，所以这些Pod的IP规划也很重要，不能有冲突。只要没有冲突，我们就可以想办法在整个Kubernetes的集群中找到它。

综上所述，要想支持不同Node上的Pod之间的通信，就要达到两个条件：

(1) 在整个Kubernetes集群中对Pod的IP分配进行规划，不能有冲突；

(2) 找到一种办法，将Pod的IP和所在Node的IP关联起来，通过这个关联让Pod可以互相访问。

根据条件1的要求，我们需要在部署Kubernetes的时候，对docker0的IP地址进行规划，保证每一个Node上的docker0地址没有冲突。我们可以在规划后手工配置到每个Node上，或者做一个分配规则，由安装的程序自己去分配占用。例如Kubernetes的网络增强开源软件Flannel就能够管理资源池的分配。

根据条件2的要求，Pod中的数据在发出时，需要有一个机制能够知道对方Pod的IP地址挂在哪个具体的Node上。也就是说先要找到Node对应宿主机的IP地址，将数据发送到这个宿主机的网卡上，然后在宿主机上将相应的数据转到具体的docker0上。一旦数据到达宿主机Node，则那个Node内部的docker0便知道如何将数据发送到Pod。如图3.26所示。

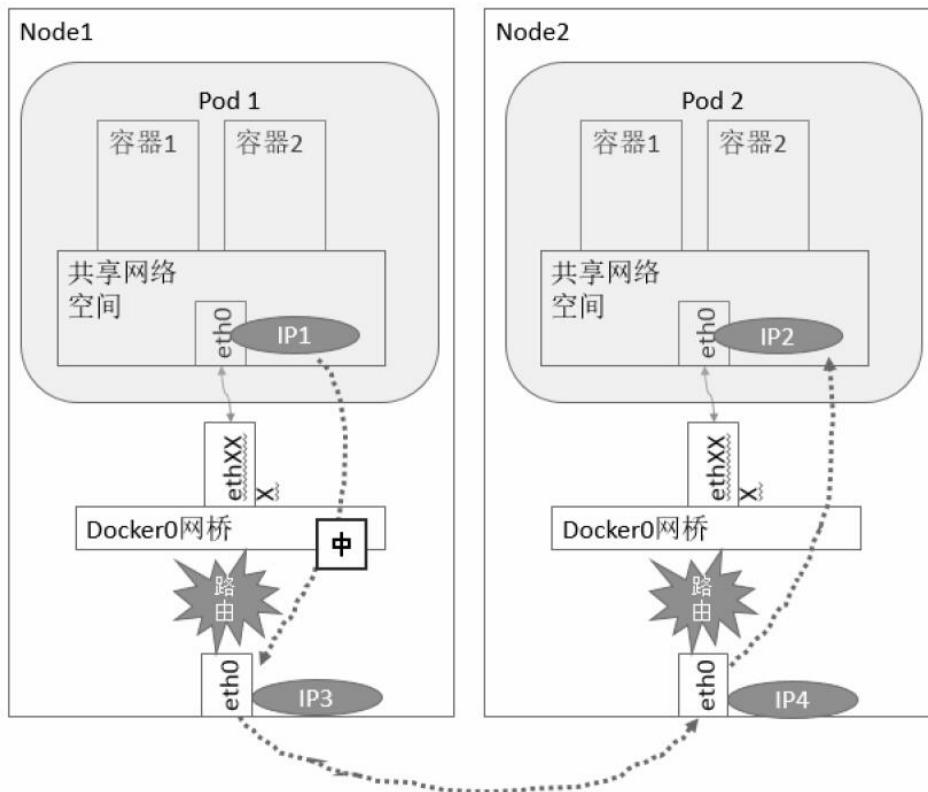


图3.26 跨Node的Pod通信

在图3.26中，IP1对应的是Pod1，IP2对应的是Pod2。Pod1在访问Pod2时，首先要将数据从源Node的eth0发送出去，找到并到达Node2的eth0。也就是说先要从IP3到IP4，之后才是IP4到IP2的递送。

在谷歌的GCE环境下，Pod的IP管理（类似docker0）、分配及它们之间的路由打通都是由GCE完成的。Kubernetes作为主要在GCE上面运行的框架，它的设计是假设底层已经具备这些条件，所以它分配完地址并将地址记录下来就完成了它的工作。在实际的GCE环境中，GCE的网络组件会读取这些信息，实现具体的网络打通。

而在实际的生产中，因为安全、费用、合规等种种原因，Kubernetes的客户不可能全部使用谷歌的GCE环境，所以在实际的私

有云环境中，除了部署Kubernetes和Docker，还需要额外的网络配置，甚至通过一些软件来实现Kubernetes对网络的要求。做到这些后，Pod和Pod之间才能无差别地透明通信。

为了达到这个目的，开源界有不少应用来增强Kubernetes、Docker的网络，在后面的章节里会介绍几个常用的组件和它们的组网原理。

3.7.5 开源的网络组件

Kubernetes的网络模型假定了所有Pod都在一个可以直接连通的扁平的网络空间中。这在GCE里面是现成的网络模型，Kubernetes假定这个网络已经存在。而在私有云里搭建Kubernetes集群，就不能假定这种网络已经存在了。我们需要自己实现这个网络假设，将不同节点上的Docker容器之间的互相访问先打通，然后运行Kubernetes。

目前已经有多个开源组件支持这个网络模型。这里介绍几个常见的模型，分别是Flannel、Open vSwitch及直接路由的方式。

1.Flannel

Flannel之所以可以搭建Kubernetes依赖的底层网络，是因为它能实现以下两点。

（1）它能协助Kubernetes，给每一个Node上的Docker容器分配互不冲突的IP地址。

（2）它能在这些IP地址之间建立一个覆盖网络（OverlayNetwork），通过这个覆盖网络，将数据包原封不动地传递到目标容器内。

通过图3.27来看看Flannel是如何实现这两点的。

可以看到，Flannel首先创建了一个名为flannel0的网桥，而且这个网桥的一端连接docker0网桥，另一端连接一个叫作flanneld的服务进程。

flanneld进程并不简单，它首先上连etcd，利用etcd来管理可分配的IP地址段资源，同时监控etcd中每个Pod的实际地址，并在内存中建立了一个Pod节点路由表；然后下连docker0和物理网络，使用内存中的Pod节点路由表，将docker0发给它的数据包包装起来，利用物理网络的连接将数据包投递到目标flanneld上，从而完成Pod到Pod之间的直接的地址通信。

Flannel之间的底层通信协议的可选余地很多，有UDP、VxLan、AWS VPC等多种方式，只要能通到对端的Flannel就可以了。源flanneld加包，目标flanneld解包，最终docker0看到的就是原始的数据，非常透明，根本感觉不到中间Flannel的存在。常用的是UDP。

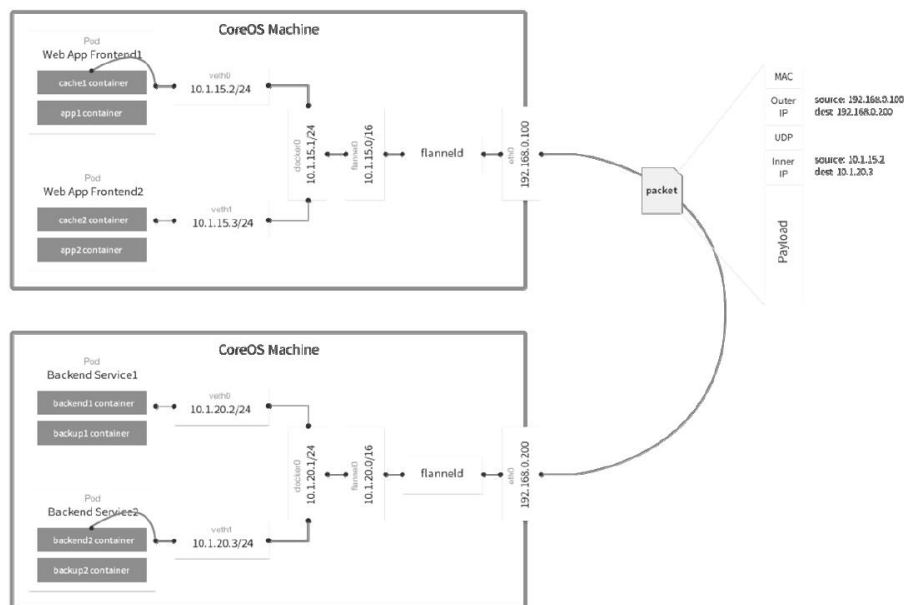


图3.27 Flannel架构图

我们看一下Flannel是如何做到让为不同Node上的Pod分配的IP不产生冲突的。其实想到Flannel使用了集中的etcd存储就很容易理解了。它每次分配的地址段都在同一个公共区域获取，这样大家自然能够互相协调，不产生冲突了。而且在Flannel分配好地址段后，后面的事情是由Docker完成的，Flannel通过修改Docker的启动参数将分配给它的地址段传递进去。

```
--bip=172.17.18.1/24
```

通过这些操作，Flannel就控制了每个Node上的docker0地址段的地址，也就保障了所有Pod的IP地址在同一个水平网络中且不产生冲突了。

Flannel完美地实现了对Kubernetes网络的支持，但是它引入了多个网络组件，在网络通信时需要转到flannel0网络接口，再转到用户态的flanneld程序，到对端后还需要走这个过程的反过程，所以也会引入一些网络的时延损耗。

另外，Flannel模型默认使用了UDP作为底层传输协议，UDP本身是非可靠协议，虽然两端的TCP实现了可靠传输，但在大流量、高并发应用场景下还需要反复测试，确保没有问题。

2.Open vSwitch

在了解了Flannel后，我们再看看Open vSwitch是怎么解决上述两个问题的。

Open vSwitch是一个开源的虚拟交换机软件，有点儿像Linux中的bridge，但是功能要复杂得多。Open vSwitch的网桥可以直接建立多种通信通道（隧道），例如Open vSwitch with GRE/VxLAN。这些通道的建立可以很容易地通过OVS的配置命令实现。在Kubernetes、Docker场景下，我们主要是建立L3到L3的隧道。举一个例子来看看Open vSwitch with GRE/VxLAN的网络架构，如图3.28所示。

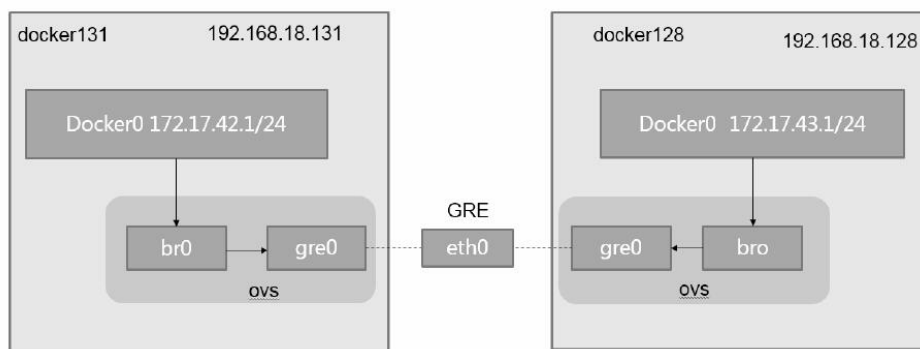


图3.28 OVS with GRE原理图

首先，为了避免Docker创建的docker0地址产生冲突（因为Docker Daemon启动且给docker0选择子网地址时只有几个备选列表，很容易产生冲突），我们可以将docker0网桥删除，手动建立一个Linux网桥，然后手动给这个网桥配置IP地址范围。

其次，建立Open vSwitch的网桥ovs，然后使用ovs-vsctl命令给ovs网桥增加gre端口，添加gre端口时要将目标连接的NodeIP地址设置为对端的IP地址。对每一个对端IP地址都需要这么操作（对于大型集群网络，这可是个体力活，要做自动化脚本来完成）。

最后将ovs的网桥作为网络接口，加入Docker的网桥上（docker0或者自己手工建立的新网桥）。

重启 ovs 网桥和 Docker 的网桥，并添加一个 Docker 的地址段到 Docker 网桥的路由规则项，就可以将两个容器的网络连接起来了。

1) 网络通信过程

当容器内的应用访问另一个容器的地址时，数据包会通过容器内的默认路由发送给 docker0 网桥。ovs 的网桥是作为 docker0 网桥的端口存在的，它会将数据发送给 ovs 网桥。ovs 网络已经通过配置建立了和其他 ovs 网桥的 GRE/VxLAN 隧道，自然能将数据送达对端的 Node，并送往 docker0 及 Pod。

通过新增的路由项，使得 Node 节点本身的应用的数据也路由到 docker0 网桥上，和刚才的通信过程一样，自然也可以访问其他 Node 上的 Pod。

2) OVS with GRE/VxLAN 组网方式的特点

OVS 的优势是，作为开源虚拟交换机软件，它相对比较成熟和稳定，而且支持各类网络隧道协议，经过了 OpenStack 等项目的考验。

另一方面，在前面介绍 Flannel 的时候可知 Flannel 除了支持建立覆盖网络（Overlay Network），保证 Pod 到 Pod 的无缝通信，还和 Kubernetes、Docker 架构体系结合紧密。Flannel 能够感知 Kubernetes 的 Service，动态维护自己的路由表，还通过 etcd 来协助 Docker 对整个 Kubernetes 集群中 docker0 的子网地址分配。而我们在使用 OVS 的时候，很多事情就需要手工完成了。

无论是 OVS 还是 Flannel，通过覆盖网络提供的 Pod 到 Pod 通信都会引入一些额外的通信开销，如果是对网络依赖特别重的应用，则需要

评估对业务的影响。

3.直接路由

我们知道，`docker0`网桥上的IP地址在Node网络上是不看到的。从一个Node到一个Node内的`docker0`是不通的。因为它不知道某个IP地址在哪里。如果能够让这些机器知道对端`docker0`地址在哪里，就可以让这些`docker0`互相通信了。这样所有Node上运行的Pod就可以互相通信了。

我们可以通过部署MultiLayer Switch（MLS）来实现这一点，在MLS中配置每个`docker0`子网地址到Node地址的路由项，通过MLS将`docker0`的IP寻址定向到对应的Node节点上。

另外，我们还可以将这些`docker0`和Node的匹配关系配置在Linux操作系统的路由项中，这样通信发起的Node能够根据这些路由信息直接找到目标Pod所在的Node，将数据传输过去。如图3.29所示。

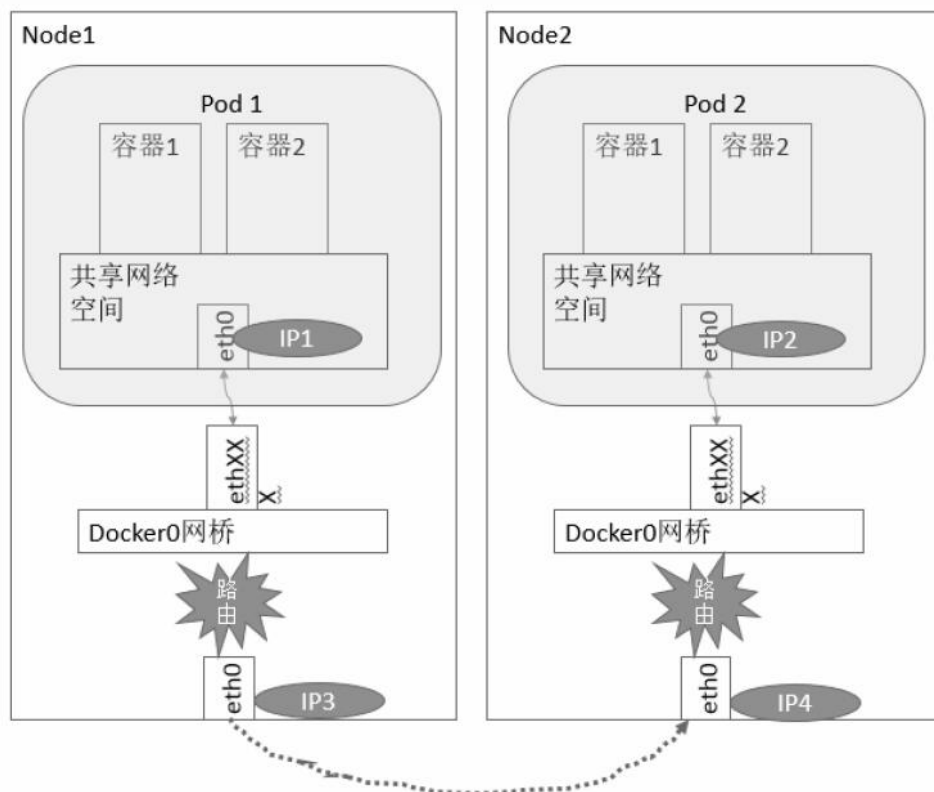


图3.29 直接路由Pod到Pod通信

我们在每个Node的路由表中增加对方所有docker0的路由项。

例如Pod1所在docker0网桥的IP子网是10.1.10.0，Node的地址为192.168.1.128；而Pod2所在docker0网桥的IP子网是10.1.20.0，Node的地址为192.168.1.129。

在Node1上用route add命令增加一条到Node2上docker0的静态路由规则：

```
route add -net 10.1.20.0 netmask 255.255.255.0 gw 192.168.1.129
```

同样，在Node2上增加一条到Node1上docker0的静态路由规则：

```
route add -net 10.1.10.0 netmask 255.255.255.0 gw  
192.168.1.128
```

这样两个Node之间的Pod就可以互相通信了，因为它们发出的数据包经过本地Linux的路由规则，能将数据送到对端的Node。

在大规模集群中，在每个Node上都需要配置到其他docker0/Node的路由项，会带来很大的工作量；并且在新增机器时，对所有Node都需要修改配置；重启机器时，如果docker0的地址有变化，则也需要修改所有Node的配置，这显然是非常复杂的。

为了管理这些动态变化的docker0地址，动态地让其他Node都感知到它，还可以使用动态路由发现协议来同步这些变化。运行动态路由发现协议代理的Node，会将本机LOCAL路由表的IP地址通过组播协议发布出去，同时监听其他Node的组播包。通过这样的信息交换，Node上的路由规则都能够相互学习到。当然，路由发现协议本身还是很复杂的，感兴趣的话你可以查阅相关的规范。在实现这些动态路由发现协议的开源软件中，常用的有Quagga、Zebra等。下面简单介绍直接路由的操作过程。

(1) 首先手工分配Docker bridge的地址，保证它们在不同的网段是不重叠的。建议最好不用Docker Daemon自动创建的docker0（因为我们不需要它的自动管理功能），而是单独建立一个bridge，给它配置规划好的IP地址，然后使用--bridge=XX来指定网桥。

(2) 然后在每一个节点上运行Quagga。

完成这些操作后，我们很快就能得到一个Pod和Pod直接互相访问的环境了。由于路由发现能够被网络上的所有设备接收，所以如果网络上的路由器也能打开RIP协议选项，则能够学习到这些路由信息。通过这些路由器，我们甚至可以在非Node节点上使用Pod的IP地址直接访问Node上的Pod。

当然，聪明的你还会有新的疑问：这样做的话，由于每一个Pod的地址都会被路由发现协议广播出去，会不会存在路由表过大的情况？实际上，路由表通常都会有高速缓存，查找速度会很快，不会对性能产生太大的影响。当然，如果你的集群容量在数千台Node以上，则仍然需要测试和评估路由表的效率问题。

3.7.6 网络实战

Docker给我们带来了不同的网络模式，而Kubernetes也以一种不同的方式来解决这些网络模式的挑战，但是其方式有些不太好理解，特别是对于刚开始接触Kubernetes的网络的开发者。我们在前面学习了Kubernetes、Docker的理论，本节将通过一个完整的实验，从部署一个Pod开始，一步一步地部署那些Kubernetes的组件，来剖析Kubernetes在网络层是如何实现及如何工作的。

这里使用虚拟机来完成实验。如果你要部署在物理机器上，或者部署在云服务商的环境下，则涉及的网络模型很可能稍微有所不同。不过，从网络角度来看，Kubernetes的机制是类似且一致的。

好了，来看看我们的实验环境，如图3.30所示。

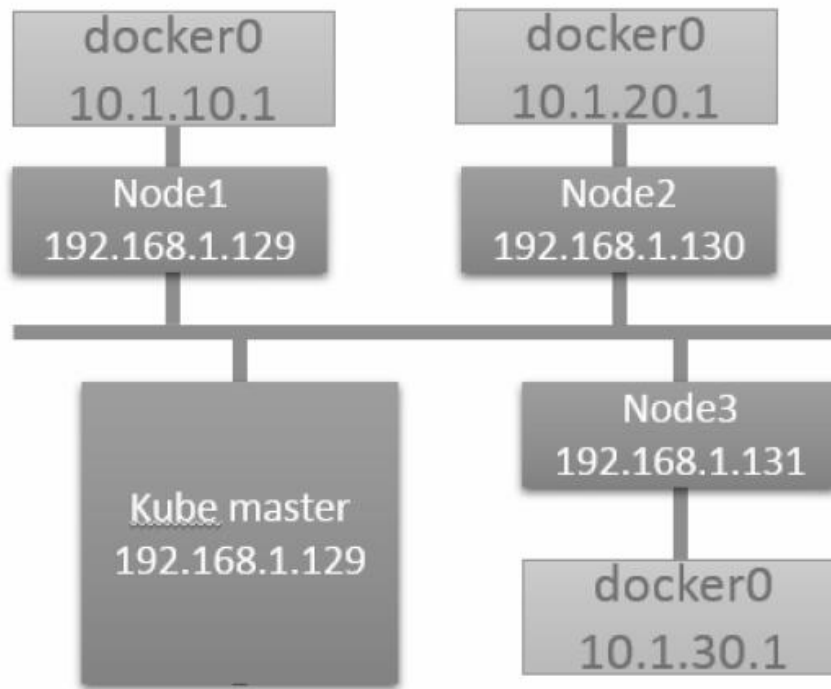


图3.30 实验环境

Kubernetes的网络模型要求每一个Node上的容器都可以相互访问。

默认的Docker的网络模型提供了一个IP地址段是172.17.0.0/16的docker0网桥。每一个容器都会在这个子网内获得IP地址，并且将docker0网桥的IP地址（172.17.42.1）作为其默认网关。需要注意的是Docker宿主机外面的网络不需要知道任何关于这个172.17.0.0/16的信息或者知道如何连接到它内部，因为Docker的宿主机针对容器发出的数据，在物理网卡地址后面都做了IP伪装MASQUERADE（隐含NAT）。也就是说，在网络上看到的任何容器数据流都来源于那台Docker节点的物理IP地址。这里所说的网络都是指连接这些主机的物理网络。

这个模型便于使用，但是并不完美，需要依赖端口映射的机制。

在Kubernetes的网络模型中，每台主机上的docker0网桥都是可以路由到的。也就是说，在部署了一个Pod的时候，在同一个集群内，那台主机的外面可以直接访问到那个Pod，并不需要在那台物理主机上做端口映射。综上所述，你可以在网络层将Kubernetes的节点看作一个路由器。如果我们将实验环境改画成一个网络图，那么它看起来如图3.31所示。

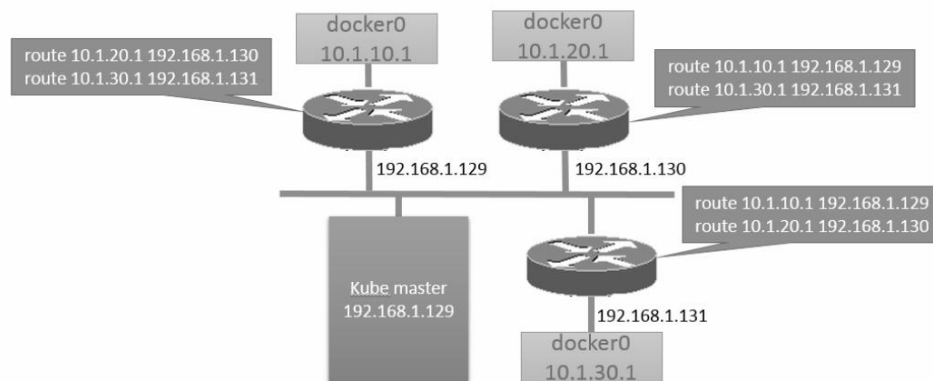


图3.31 实验环境网络图

为了支持Kubernetes网络模型，我们采取了直接路由的方式来实现，在每个Node上配置相应的静态路由项，例如在192.168.1.129这个Node上我们配置了两个路由项：

```
# route add -net 10.1.20.0 netmask 255.255.255.0 gw
192.168.130

# route add -net 10.1.30.0 netmask 255.255.255.0 gw
192.168.131
```

这意味着，每一个新部署的容器都将使用这个Node（docker0的网桥IP）作为它的默认网关。而这些Node节点（类似路由器）都有其他docker0的路由信息，这样它们就能够相互连通了。

接下来通过一些实际的案例，来看看Kubernetes在不同的场景下其网络部分到底做了什么事情。

第1步：部署一个RC/Pod

部署的RC/Pod描述文件如下（frontend-controller.yaml）：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  replicas: 1
  selector:
    name: frontend
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
        - name: php-redis
```

```

    image: kubeguide/guestbook-php-frontend
    env:
      - name: GET_HOSTS_FROM
        value: env
    ports:
      - containerPort: 80
        hostPort: 80

```

为了便于观察，我们假定在一个空的Kubernetes集群上运行，提前清理了所有Replication Controller、Pod和其他Service：

```

# kubectl get rc

```

	CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR	REPLICAS

```

#
# kubectl get services

```

	NAME	LABELS
SELECTOR	IP(S)	PORT(S)
	kubernetes	component=apiserver,provider=kubernetes
<none>	20.1.0.1	443/TCP

```

#
# kubectl get pods

```

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

让我们检查一下此时某个Node上的网络接口都有哪些。Node1的状态是：

```
# ifconfig
    docker0: flags=4099<UP,BROADCAST,RUNNING,MULTICAST>
mtu 1500
            inet 10.1.10.1    netmask 255.255.255.0
broadcast 10.1.10.255
            inet6 fe80::5484:7aff:fefe:9799    prefixlen 64
scopeid 0x20<link>
            ether 56:84:7a:fe:97:99    txqueuelen 0
(Ethernet)
    RX packets 373245    bytes 170175373 (162.2 MiB)
    RX errors 0    dropped 0    overruns 0    frame 0
    TX packets 353569    bytes 353948005 (337.5 MiB)
    TX errors 0    dropped 0    overruns 0    carrier 0
collisions 0

                                                    eno16777736:
flags=4163<UP,BROADCAST,RUNNING,MULTICAST>    mtu 1500
            inet 192.168.1.129    netmask 255.255.255.0
broadcast 192.168.1.255
            inet6 fe80::20c:29ff:fe47:6e2c    prefixlen 64
scopeid 0x20<link>
            ether 00:0c:29:47:6e:2c    txqueuelen 1000
(Ethernet)
    RX packets 326552    bytes 286033393 (272.7 MiB)
    RX errors 0    dropped 0    overruns 0    frame 0
    TX packets 219520    bytes 31014871 (29.5 MiB)
```

```
TX errors 0 dropped 0 overruns 0 carrier 0
collisions 0
```

```
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 0 (Local Loopback)
    RX packets 24095 bytes 2133648 (2.0 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 24095 bytes 2133648 (2.0 MiB)
    TX errors 0 dropped 0 overruns 0 carrier 0
collisions 0
```

可以看出，有一个docker0网桥和一个本地地址的网络端口。现在部署一下我们在前面准备的RC/Pod配置文件，看看发生了什么：

```
# kubectl create -f frontend-controller.yaml
replicationcontrollers/frontend
#
# kubectl get pods
```

	NAME	READY	STATUS	RESTARTS
AGE	NODE			
	frontend-4o11g	1/1	Running	0
	192.168.1.130			11s

可以看到一些有趣的事情。Kubernetes为这个Pod找了一个主机192.168.1.130（Node2）来运行它。另外，这个Pod还获得了一个在

Node2的docker0网桥上的IP地址。我们登录到Node2上看看发生了什么事情：

```
# docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
37b193a4c633	kubeguide/example-guestbook-php-redis	"/bin/sh -c /run.sh"	32 seconds ago
Up 26 seconds	k8s_php-redis.6ad3289e_frontend-n9n1m_development_813e2dd9-8149-11e5-823b-000c2921ba71_af6dd859		
6d1b99cff4ae	google_containers/pause:latest	"/pause"	35 seconds ago
Up 28 seconds	0.0.0.0:80->80/tcp	k8s_POD.855eeb3d_frontend-4t52y_development_813e3870-8149-11e5-823b-000c2921ba71_2b66f05e	

在Node2上现在运行了两个容器。在我们的RC/Pod定义文件中仅仅包含了一个，那么这第2个是从哪里来的呢？第2个看起来运行的是一个叫作google_containers/pause: latest的镜像，而且这个容器已经有端口映射到它上面了，为什么是这样呢？让我们深入容器内部去看一下具体原因。使用Docker的“inspect”命令来查看容器的详细信息，特别要关注容器的网络模型。

```
# docker inspect 6d1b99cff4ae | grep NetworkMode
    "NetworkMode": "bridge",
# docker inspect 37b193a4c633 | grep NetworkMode
```

```
"NetworkMode":  
"container:6d1b99cff4ae537689ce87d7528f4ba9dbb40ae711ecc0a5  
b3f7c39ff5e5e495",
```

有趣的结果是，在查看完每个容器的网络模型后，我们可以看到这样的配置：我们检查的第1个容器是运行了“google_containers/pause: latest”镜像的容器，它使用了Docker默认的网络模型bridge；而我们检查的第2个容器，也就是在我们RC/Pod中定义运行的php-redis容器，使用了非默认的网络配置和映射容器的模型，指定了映射目标容器为“google_containers/pause: latest”。

我们一起来仔细思考一下这个过程，为什么Kubernetes要这么做呢？首先，一个Pod内的所有容器都需要共用同一个IP地址，这就意味着一定要使用网络的容器映射模式。然而，为什么不能只启动第1个Pod中的容器，而将第2个Pod内的容器关联到第1个容器呢？我们认为Kubernetes从两个方面来考虑这个问题：首先，如果Pod有超过两个容器的话，则连接这些容器可能不容易；其次，后面的容器还要依赖第1个被关联的容器，如果第2个容器关联到第1个容器，且第1个容器死掉的话，第2个也将死掉。启动一个基础容器，然后将Pod内的所有容器都连接到它上面会更容易一些。因为我们只需要为基础的这个Google_containers/pause容器执行端口映射规则，这也简化了端口映射的过程。所以我们的Pod的网络模型类似于图3.32。

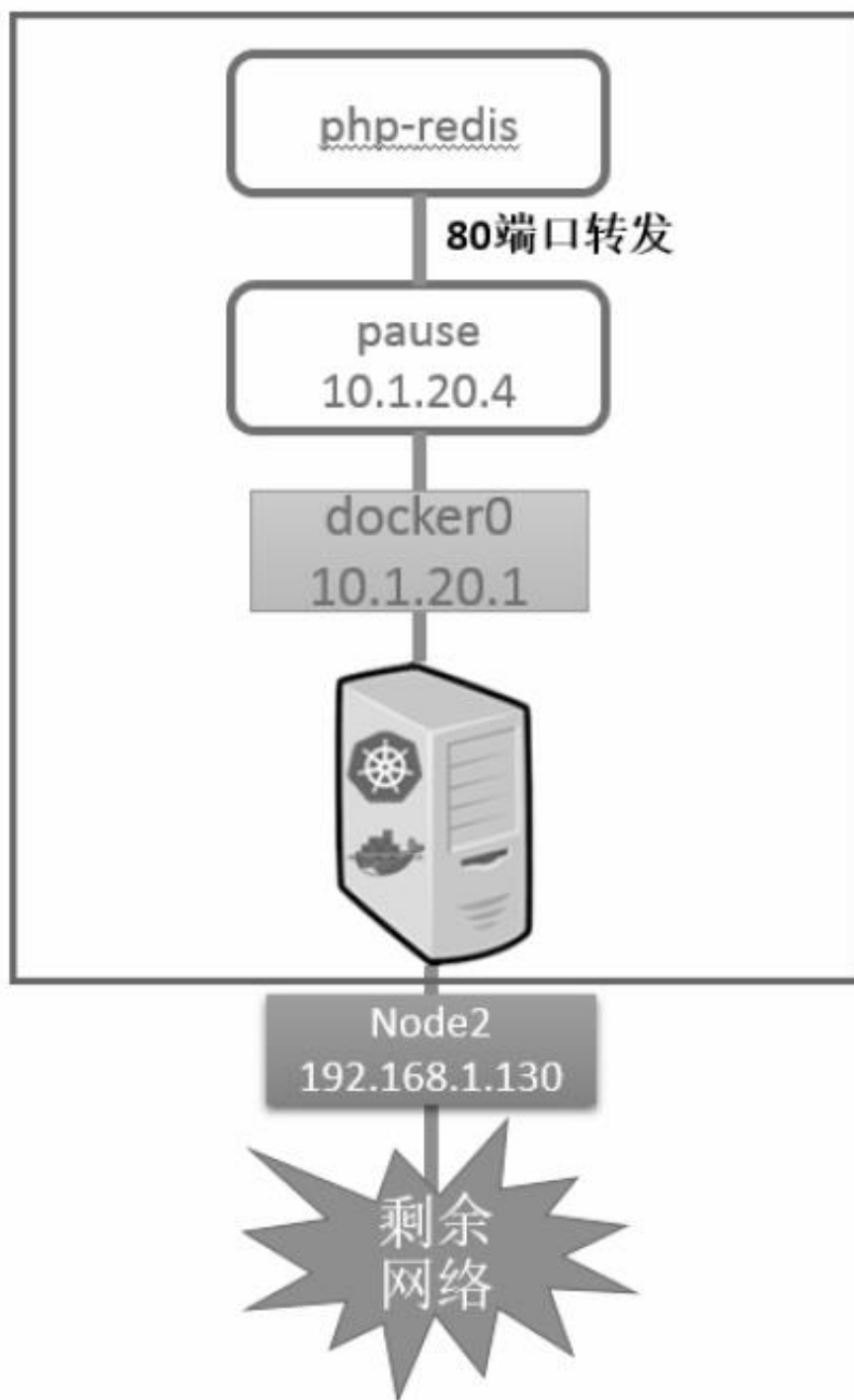


图3.32 启动Pod后网络模型

在这种情况下，实际Pod的IP数据流的网络目标都是这个google_containers/pause容器。图3.32有点儿取巧地显示了是google_containers/pause容器将端口80的流量转发给了相关的容器。而Pause只是逻辑上的，并没有真的这么做。实际上另外的Web容器直接监听了这些端口，和google_containers/pause容器共享了同一个网络堆栈。这就是为什么Pod内部实际容器的端口映射都显示到google_containers/pause容器上了。我们可以通过docker port命令来检验一下：

```
# docker ps
CONTAINER ID        IMAGE
37b193a4c633      kubeguide/example-guestbook-php-redis
6d1b99cff4ae      google_containers/pause:latest
#
# docker port 6d1b99cff4ae
80/tcp ->0.0.0.0:80
```

综上所述，google_containers/pause容器实际上只是负责接管这个Pod的Endpoint，它实际上并没有做更多的事情。那么Node呢，它需要将数据流传给google_containers/pause容器吗？我们来检查一下Iptables的规则，看看有什么发现：

```
# iptables-save
# Generated by iptables-save v1.4.21 on Thu Sep 24
17:15:01 2015
*nat
```

```

:PREROUTING ACCEPT [0:0]
:INPUT ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]
:DOCKER - [0:0]
:KUBE-NODEPORT-CONTAINER - [0:0]
:KUBE-NODEPORT-HOST - [0:0]
:KUBE-PORTALS-CONTAINER - [0:0]
:KUBE-PORTALS-HOST - [0:0]

-A PREROUTING -m comment --comment "handle ClusterIPs;
NOTE: this must be before the NodePort rules" -j KUBE-
PORTALS-CONTAINER

-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER

-A PREROUTING -m addrtype --dst-type LOCAL -m comment
--comment "handle service NodePorts; NOTE: this must be the
last rule in the chain" -j KUBE-NODEPORT-CONTAINER

-A OUTPUT -m comment --comment "handle ClusterIPs;
NOTE: this must be before the NodePort rules" -j KUBE-
PORTALS-HOST

-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type
LOCAL -j DOCKER

-A OUTPUT -m addrtype --dst-type LOCAL -m comment --
comment "handle service NodePorts; NOTE: this must be the
last rule in the chain"

-A POSTROUTING -s 10.1.20.0/24 ! -o docker0 -j
MASQUERADE

-A KUBE-PORTALS-CONTAINER -d 20.1.0.1/32 -p tcp -m

```

```

comment --comment "default/kubernetes:" -m tcp --dport 443
-j REDIRECT --to-ports 60339
    -A KUBE-PORTALS-HOST -d 20.1.0.1/32 -p tcp -m comment
--comment "default/kubernetes:" -m tcp --dport 443 -j DNAT
--to-destination 192.168.1.131:60339
COMMIT
# Completed on Thu Sep 24 17:15:01 2015
# Generated by iptables-save v1.4.21 on Thu Sep 24
17:15:01 2015
*filter
:INPUT ACCEPT [1131:377745]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [1246:209888]
:DOCKER - [0:0]
-A FORWARD -o docker0 -j DOCKER
    -A FORWARD -o docker0 -m conntrack --ctstate
RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A DOCKER -d 172.17.0.19/32 ! -i docker0 -o docker0 -p
tcp -m tcp --dport 5000 -j ACCEPT
COMMIT
# Completed on Thu Sep 24 17:15:01 2015

```

上面的这些规则并没有应用到我们刚刚定义的Pod。当然，Kubernetes会给每一个Kubernetes的节点提供一些默认的服务，上面的规则就是Kubernetes的默认服务需要的。关键是，我们没有看到任何IP

伪装的规则，并且没有任何指向Pod 10.1.20.4的内部方向的端口映射。

第2步：发布一个服务

我们已经了解了Kubernetes如何处理最基本的元素Pod的连接问题，接下来看一下它是如何处理Service的。Service允许我们在多个Pod之间抽象一些服务，而且，服务可以通过提供在同一个Service的多个Pod之间的负载均衡机制来支持水平扩展。我们再次将环境初始化，删除刚刚创建的RC/Pod来确保集群是空的：

```
# kubectl stop rc frontend
replicationcontroller/frontend
#
# kubectl get rc
CONTROLLER      CONTAINER(S)      IMAGE(S)      SELECTOR
REPLICAS
#
# kubectl get services
NAME                                LABELS
SELECTOR  IP(S)      PORT(S)
kubernetes  component=apiserver,provider=kubernetes
<none>20.1.0.1  443/TCP
#
# kubectl get pods
NAME      READY    STATUS    RESTARTS    AGE
```

然后准备一个名称为frontend的Service配置文件：

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  ports:
    - port: 80
#   nodePort: 30001
  selector:
    name: frontend
# type:
#   NodePort
```

然后在Kubernetes集群中定义这个服务:

```
# kubectl create -f frontend-service.yaml
services/frontend
# kubectl get services
```

	NAME	LABELS	SELECTOR
IP(S)		PORT(S)	
	frontend	name=frontend	name=frontend
20.1.244.75		80/TCP	
			kubernetes
component=apiserver,provider=kubernetes		<none>	20.1.0.1
443/TCP			

服务正确创建后，可以看到Kubernetes集群已经为这个服务分配了一个虚拟IP地址20.1.244.75，这个IP地址是在Kubernetes的Portal Network中分配的。而这个Portal Network的地址范围则是我们在Kubmaster上启动API服务进程时，使用--service-cluster-ip-range=xx命令行参数指定的：

```
# cat /etc/kubernetes/apiserver
.....
# Address range to use for services
    KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=20.1.0.0/16"
.....
```

这个IP段可以是任何段，只要不和docker0或者物理网络的子网冲突就可以。选择任意其他网段的原因是这个网段将不会在物理网络和docker0网络上进行路由。这个Portal Network针对每一个Node都有局部的特殊性，实际上它存在的意义是让容器的流量都指向默认网关（也就是docker0网桥）。在继续实验前，先登录到Node1上看一下我们定义服务后发生了什么变化。首先检查一下Iptables/Netfilter的规则：

```
# iptables-save
.....
-A KUBE-PORTALS-CONTAINER -d 20.1.244.75/32 -p tcp -m comment --comment "default/frontend:" -m tcp --dport 80 -j REDIRECT --to-ports 59528
-A KUBE-PORTALS-HOST -d 20.1.244.75/32 -p tcp -m
```

```
comment --comment "default/kubernetes:" -m tcp --dport 80 -  
j DNAT --to-destination 192.168.1.131:59528
```

.....

第1行是挂在PREROUTING链上的端口重定向规则，所有的进流量如果满足20.1.244.75: 80，则都会被重定向到端口33761。第2行是挂在OUTPUT链上的目标地址NAT，做了和上述第1行规则类似的工作，但针对的是当前主机生成的外出流量。所有主机生成的流量都需要使用这个DNAT规则来处理。简而言之，这两个规则使用了不同的方式做了类似的事情，就是将所有从节点生成的发送给20.1.244.75: 80的流量重定向到本地的33761端口。

到此为止，目标为Service IP地址和端口的任何流量都将被重定向到本地的33761端口。这个端口连到哪里去了呢？这就到了kube-proxy发挥作用的地方了。这个kube-proxy服务给每一个新创建的服务关联了一个随机的端口号，并且监听那个特定的端口，为服务创建相关的负载均衡对象。在我们的实验中，随机生成的端口刚好是33761。通过监控Node1上的Kubernetes-Service的日志，在创建服务时，我们可以看到下面的记录：

```
2612 proxier.go:413] Opened iptables from-containers  
portal for service "default/frontend:"on TCP 20.1.244.75:80  
2612 proxier.go:424] Opened iptables from-host portal  
for service "default/frontend:"on TCP 20.1.244.75:80
```

现在我们知道，所有的流量都被导入kube-proxy。现在我们需要它完成一些负载均衡的工作。创建Replication Controller并观察结果，

下面是Replication Controller的配置文件:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  replicas: 3
  selector:
    name: frontend
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
        - name: php-redis
          image: kubeguide/example-guestbook-php-redis
          env:
            - name: GET_HOSTS_FROM
              value: env
          ports:
            - containerPort: 80
#           hostPort: 80
```

在集群发布上述配置文件后，等待并观察，确保所有Pod都运行起来了：

```
# kubectl create -f frontend-controller.yaml
replicationcontrollers/frontend
#
# kubectl get pods -o wide
```

	NAME	READY	STATUS	RESTARTS	AGE
NODE					
	frontend-64t8q	1/1	Running	0	5s
192.168.1.130					
	frontend-dzqve	1/1	Running	0	5s
192.168.1.131					
	frontend-x5dwy	1/1	Running	0	5s
192.168.1.129					

现在所有的Pod都运行起来了，Service将会对匹配到标签为“name=frontend”的所有Pod进行负载分发。因为Service的选择匹配所有的这些Pod，所以我们的负载均衡将会对这3个Pod进行分发。现在我们做实验的环境如图3.33所示。

```
yum-yinstall tcpdump
```

安装完成后，登录Node1，运行tcpdump命令：

```
tcpdump-nn-q-ieno16777736 port80
```

需要捕获物理服务器以太网接口的数据包，Node1机器上的以太网接口名字叫作eno16777736。

再打开第1个窗口运行第2个tcpdump程序，不过我们需要一些额外的信息去运行它，即挂接在docker0桥上的虚拟网卡Veth的名字。我们看到只有一个frontend容器在Node1主机上运行，所以可以使用简单的“ip addr”命令来查看唯一的“Veth”网络接口：

```
# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd
00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eno16777736: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu
1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:47:6e:2c brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.129/24 brd 192.168.1.255 scope
```

```
global eno16777736
    valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe47:6e2c/64 scope link
    valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu
1500 qdisc noqueue state DOWN
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
    inet 10.1.10.1/24 brd 10.1.10.255 scope global
docker0
    valid_lft forever preferred_lft forever
    inet6 fe80::5484:7aff:fefe:9799/64 scope link
    valid_lft forever preferred_lft forever
12: veth0558bfa: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu
1500 qdisc noqueue master docker0 state UP
    link/ether 86:82:e5:c8:5a:9a brd ff:ff:ff:ff:ff:ff
    inet6 fe80::8482:e5ff:fec8:5a9a/64 scope link
    valid_lft forever preferred_lft forever
```

复制这个接口的名字，在第2个窗口中运行tcpdump命令。

```
tcpdump -nn -q -i veth0558bfa host 20.1.244.75
```

同时运行这两个命令，并且将窗口并排放置，以便同时看到两个窗口的输出：

```
# tcpdump -nn -q -i eno16777736 port 80
tcpdump: verbose output suppressed, use -v or -vv for
```

```
full protocol decode
    listening on eno16777736, link-type EN10MB (Ethernet),
capture size 65535 bytes

# tcpdump -nn -q -i veth0558bfa host 20.1.244.75
tcpdump: verbose output suppressed, use -v or -vv for
full protocol decode
    listening on veth0558bfa, link-type EN10MB (Ethernet),
capture size 65535 bytes
```

好了，我们已经在同时捕获两个接口的网络包了。这时再启动第3个窗口，运行一个“**docker exec**”命令来连接到我们的“**frontend**”的容器内部（你可以先执行**docker ps**来获得这个容器的ID）：

```
# docker ps
```

	CONTAINER	ID	IMAGE
.....			
	268ccdfb9524		kubeguide/example-guestbook-php-
redis		
	6a519772b27e		google_containers/pause:latest
.....			

执行命令进入容器内部：

```
#docker exec -it 268ccdfb9524 bash
# docker exec -it 268ccdfb9524 bash
root@frontend-x5dwy:/#
```

一旦进入运行的容器内部，我们就可以通过Pod的IP地址来访问服务了。使用curl来尝试访问服务：

```
curl20.1.244.75
```

在使用curl访问服务时，将在抓包的两个窗口内看到：

```
20:19:45.208948 IP 192.168.1.129.57452
>10.1.30.8.8080: tcp 0
20:19:45.209005 IP 10.1.30.8.8080 >
192.168.1.129.57452: tcp 0
20:19:45.209013 IP 192.168.1.129.57452
>10.1.30.8.8080: tcp 0
20:19:45.209066 IP 10.1.30.8.8080 >
192.168.1.129.57452: tcp 0

20:19:45.209227 IP 10.1.10.5.35225 > 20.1.244.75.80:
tcp 0
20:19:45.209234 IP 20.1.244.75.80 >10.1.10.5.35225:
tcp 0
20:19:45.209280 IP 10.1.10.5.35225 > 20.1.244.75.80:
tcp 0
20:19:45.209336 IP 20.1.244.75.80 >10.1.10.5.35225:
tcp 0
```

这些信息说明了什么问题呢？让我们在网络图上用实线标出第1个窗口中网络抓包信息的含义（物理网卡上的网络流量），并用虚线标

出第2个窗口中网络抓包信息的含义（docker0网桥上的网络流量），如图3.34所示。

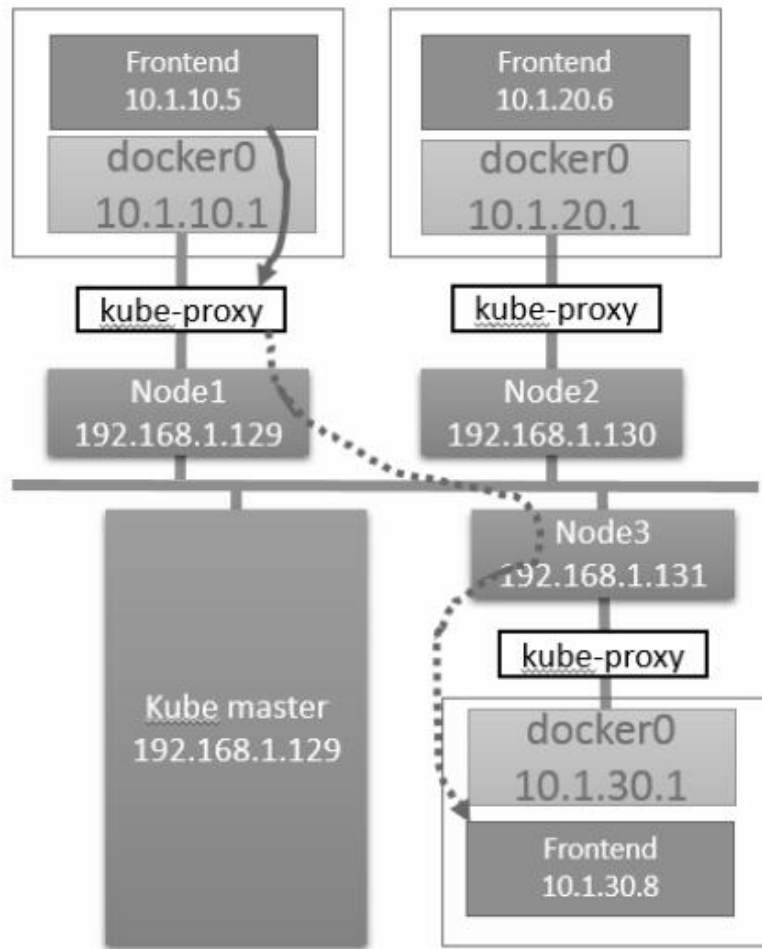


图3.34 数据流动情况图1

注意，图3.34中，虚线绕过了Node3的kube-proxy，这么做是因为Node3上的kube-proxy没有参与这次网络交互。换句话说，Node1的kube-proxy服务和负载均衡到的Pod进行网络交互。

在查看第2个捕获包的窗口时，我们能够站在容器的视角看这些流量。首先，容器尝试使用20.1.244.75: 80打开TCP的Socket连接。同

时，我们还可以看到从服务地址20.1.244.75返回的数据。从容器的视角来看，整个交互过程都是在服务之间进行的。但是在查看一个捕获包的窗口时（上面的窗口），我们可以看到物理机之间的数据交互，可以看到一个TCP连接从Node1的物理地址（192.168.1.129）发出，直接连接到运行Pod的主机Node3（192.168.1.131）。总而言之，Kubernetes的kube-proxy作为一个全功能的代理服务器管理了两个独立的TCP连接：一个是从容器到kube-proxy：另一个是从kube-proxy到负载均衡的目标Pod。

如果我们清理一下捕获的记录，再次运行curl，则还可以看到网络流量被负载均衡转发到另一个节点Node2上了。

```
                20:19:45.208948    IP    192.168.1.129.57485
>10.1.20.6.8080: tcp 0
                20:19:45.209005    IP    10.1.20.6.8080    >
192.168.1.129.57485: tcp 0
                20:19:45.209013    IP    192.168.1.129.57485
>10.1.20.6.8080: tcp 0
                20:19:45.209066    IP    10.1.20.6.8080    >
192.168.1.129.57485: tcp 0

                20:19:45.209227  IP  10.1.10.5.38026> 20.1.244.75.80:
tcp 0
                20:19:45.209234  IP  20.1.244.75.80  >10.1.10.5.38026:
tcp 0
                20:19:45.209280  IP  10.1.10.5.38026> 20.1.244.75.80:
tcp 0
```

20:19:45.209336 IP 20.1.244.75.80 >10.1.10.5.38026:

tcp 0

这一次，Kubernetes的Proxy将选择运行在Node2（10.1.20.1）上面的Pod作为负载均衡的目的。网络流动图如图3.35所示。

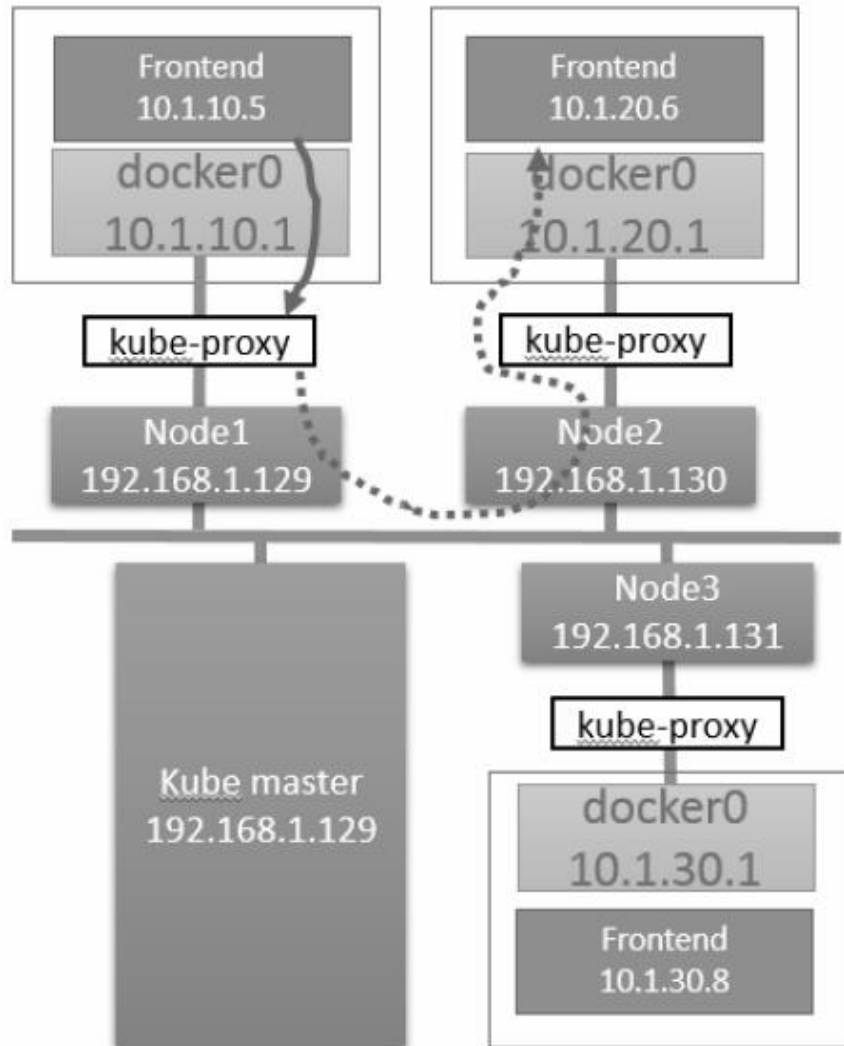


图3.35 数据流动情况图2

到这里，你肯定已经知道另外一个可能的负载均衡的路由结果了吧。

第4章 Kubernetes开发指南

本章将引入REST的概念，详细说明Kubernetes API，并举例说明如何基于Jersey和Fabric8框架访问Kubernetes API，深入分析基于这两个框架访问Kubernetes API的优缺点。下面从REST开始说起。

4.1 REST简述

REST (Representational State Transfer) 是由Roy Thomas Fielding博士在他的论文Architectural Styles and the Design of Network-based Software Architectures中提出的一个术语。REST本身只是为分布式超媒体系统设计的一种架构风格，而不是标准。

基于Web的架构实际上就是各种规范的集合，这些规范共同组成了Web架构，比如HTTP、客户端服务器模式都是规范。每当我们在原有规范的基础上增加新的规范时，就会形成新的架构。而REST正是这样一种架构，它结合了一系列规范，形成了一种新的基于Web的架构风格。

传统的Web应用大多是B/S架构，涉及如下规范。

(1) 客户-服务器：这种规范的提出，改善了用户接口跨多个平台的可移植性，并且通过简化服务器组件，改善了系统的可伸缩性。最为关键的是通过分离用户接口和数据存储，使得不同的用户终端共享相同的数据成为可能。

(2) 无状态性：无状态性是在客户-服务器约束的基础上添加的又一层规范，它要求通信必须在本质上是无状态的，即从客户端到服务器的每个request都必须包含理解该request所必需的所有信息。这个规范改善了系统的可见性（无状态性使得客户端和服务端不必保存对方的详细信息，服务器只需要处理当前的request，而不必了解所有request的历史）、可靠性（无状态性减少了服务器从局部错误中恢复

的任务量）、可伸缩性（无状态性使得服务器端可以很容易地释放资源，因为服务器端不必在多个request中保存状态）。同时，这种规范的缺点也是显而易见的，由于不能将状态数据保存在服务器上，因此增加了在一系列request中发送重复数据的开销，严重降低了效率。

（3）缓存：为了改善无状态性带来的网络的低效性，我们添加了缓存约束。缓存约束允许隐式或显式地标记一个response中的数据，赋予了客户端缓存response数据的功能，这样就可以为以后的request共用缓存的数据，部分或全部地消除一部分交互，提高了网络效率。但是由于客户端缓存了信息，所以增加了客户端与服务器数据不一致的可能性，从而降低了可靠性。

B/S架构的优点是部署非常方便，在用户体验方面却不很理想。为了改善这种情况，我们引入了REST。REST在原有架构上增加了三个新规范：统一接口、分层系统和按需代码。

（1）统一接口：REST架构风格的核心特征就是强调组件之间有一个统一的接口，表现为在REST世界里，网络上的所有事物都被抽象为资源，REST通过通用的链接器接口对资源进行操作。这样设计的好处是保证系统提供的服务都是解耦的，极大地简化了系统，从而改善了系统的交互性和可重用性。

（2）分层系统：分层系统规则的加入提高了各种层次之间的独立性，为整个系统的复杂性设置了边界，通过封装遗留的服务，使新的服务器免受遗留客户端的影响，也提高了系统的可伸缩性。

（3）按需代码：REST允许对客户端功能进行扩展。比如，通过下载并执行applet或脚本形式的代码来扩展客户端的功能。但这在改善

系统可扩展性的同时降低了可见性，所以它只是**REST**的一个可选约束。

REST架构是针对**Web**应用而设计的，其目的是为了降低开发的复杂性，提高系统的可伸缩性。**REST**提出了如下设计准则。

- (1) 网络上的所有事物都被抽象为资源 (**Resource**) 。
- (2) 每个资源对应一个唯一的资源标识符 (**Resource Identifier**) 。
- (3) 通过通用的连接器接口 (**Generic Connector Interface**) 对资源进行操作。
- (4) 对资源的各种操作不会改变资源标识符。
- (5) 所有的操作都是无状态的 (**Stateless**) 。

REST中的资源所指的不是数据，而是数据和表现形式的组合，比如“最新访问的10位会员”和“最活跃的10位会员”在数据上可能有重叠或者完全相同，而由于它们的表现形式不同，所以被归为不同的资源，这也就是为什么**REST**的全名是**Representational State Transfer**。资源标识符就是**URI** (**Uniform Resource Identifier**)，不管是图片、**Word**还是视频文件，甚至只是一种虚拟的服务，也不管是**xml**、**txt**还是其他文件格式，全部通过**URI**对资源进行唯一标识。

REST是基于**HTTP**的，任何对资源的操作行为都通过**HTTP**来实现。以往的**Web**开发大多数用的是**HTTP**中的**GET**和**POST**方法，很少使用其他方法，这实际上是因为对**HTTP**的片面理解造成的。**HTTP**不仅仅是一个简单的运载数据的协议，而且是一个具有丰富内涵的网络

软件的协议，它不仅能对互联网资源进行唯一定位，还能告诉我们如何对该资源进行操作。HTTP把对一个资源的操作限制在4种方法内：GET、POST、PUT和DELETE，这正是对资源CRUD操作的实现。由于资源和URI是一一对应的，在执行这些操作时URI没有变化，和以往的Web开发有很大的区别，所以极大地简化了Web开发，也使得URI可以被设计成更为直观地反映资源的结构。这种URI的设计被称作RESTful的URI，为开发人员引入了一种新的思维方式：通过URL来设计系统结构。当然了，这种设计方式对于一些特定情况也是不适用的，也就是说不是所有URI都适用于RESTful。

REST之所以可以提高系统的可伸缩性，就是因为它要求所有操作都是无状态的。由于没有了上下文（Context）的约束，做分布式和集群时就更为简单，也可以让系统更为有效地利用缓冲池（Pool），并且由于服务器端不需要记录客户端的一系列访问，也就减少了服务器端的性能损耗。

Kubernetes API也符合RESTful规范，下面对其进行介绍。

4.2 Kubernetes API详解

4.2.1 Kubernetes API概述

Kubernetes API是集群系统中的重要组成部分，Kubernetes中各种资源（对象）的数据通过该API接口被提交到后端的持久化存储（etcd）中，Kubernetes集群中的各部件之间通过该API接口实现解耦合，同时Kubernetes集群中一个重要且便捷的管理工具kubectl也是通过访问该API接口实现其强大的管理功能的。Kubernetes API中的资源对象都拥有通用的元数据，资源对象也可能存在嵌套现象，比如在一个Pod里面嵌套多个Container。创建一个API对象是指通过API调用创建一条有意义的记录，该记录一旦被创建，Kubernetes将确保对应的资源对象会被自动创建并托管维护。

在Kubernetes系统中，大多数情况下，API定义和实现都符合标准的HTTP REST格式，比如通过标准的HTTP动词（POST、PUT、GET、DELETE）来完成对相关资源对象的查询、创建、修改、删除等操作。但同时Kubernetes也为某些非标准的REST行为实现了附加的API接口，例如Watch某个资源的变化、进入容器执行某个操作等。另外，某些API接口可能违背严格的REST模式，因为接口不是返回单一的JSON对象，而是返回其他类型的数据，比如JSON对象流（Stream）或非结构化的文本日志数据等。

Kubernetes开发人员认为，任何成功的系统都会经历一个不断成长和不断适应各种变更的过程。因此，他们期望Kubernetes API是不断变更和增长的。同时，他们在设计和开发时，有意识地兼容了已存在的客户需求。通常，新的API资源（Resource）和新的资源域不希望被频繁地加入系统。资源或域的删除需要一个严格的审核流程。

为了方便查阅API接口的详细定义，Kubernetes使用了swagger-ui提供API在线查询功能，其官网为http://kubernetes.io/third_party/swagger-ui/，Kubernetes开发团队会定期更新、生成UI及文档。Swagger UI是一款RESTAPI文档在线自动生成和功能测试软件，关于Swagger的内容请访问官网<http://swagger.io>。

运行在Master节点上的API Server进程同时提供了swagger-ui的访问地址：<http://<master-ip>:<master-port>/swagger-ui/>。假设我们的API Server安装在192.168.1.128服务器上，绑定了8080端口，则可以通过访问<http://192.168.1.128:8080/swagger-ui/>来查看API信息，如图4.1所示。



图4.1 swagger-ui

单击api/v1可以查看所有API的列表，如图4.2所示。

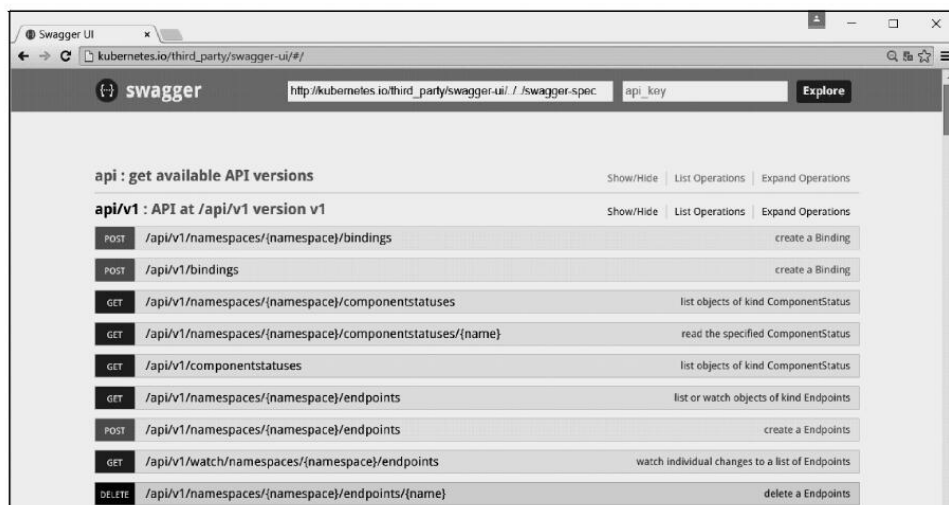


图4.2 查看API列表

以 create a Pod 为例，找到 Rest API 的访问路径为：`/api/v1/namespaces/{namespace}/pods`，如图4.3所示。

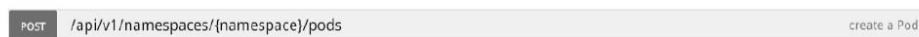


图4.3 Create a Pod API

单击链接展开，即可查看详细的API接口说明，如图4.4所示。

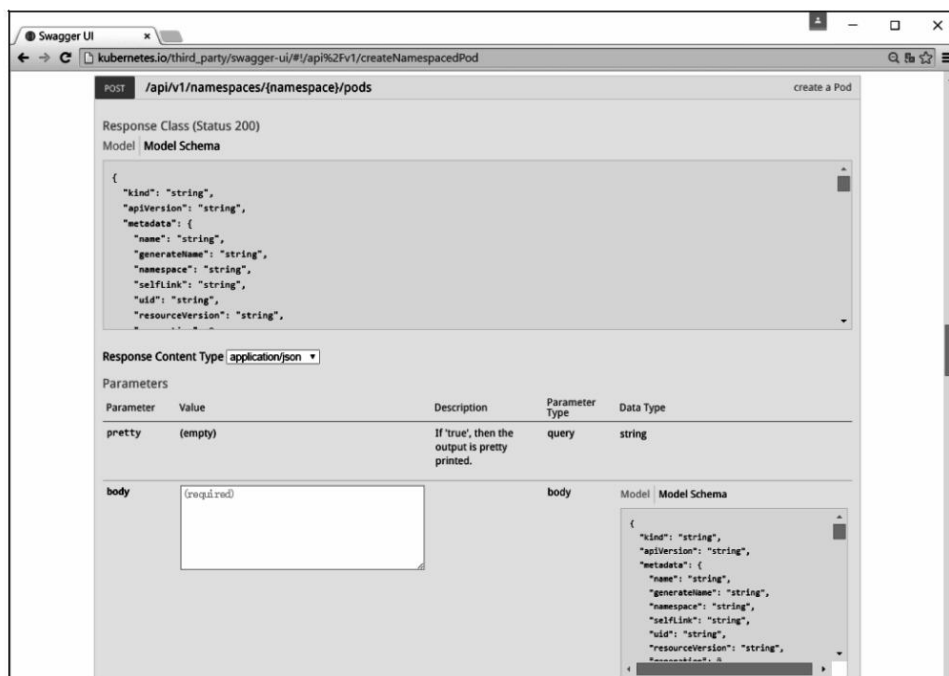


图4.4 Create a Pod API详细说明

单击Model链接，则可以查看文本格式显示的API接口描述，如图4.5所示。

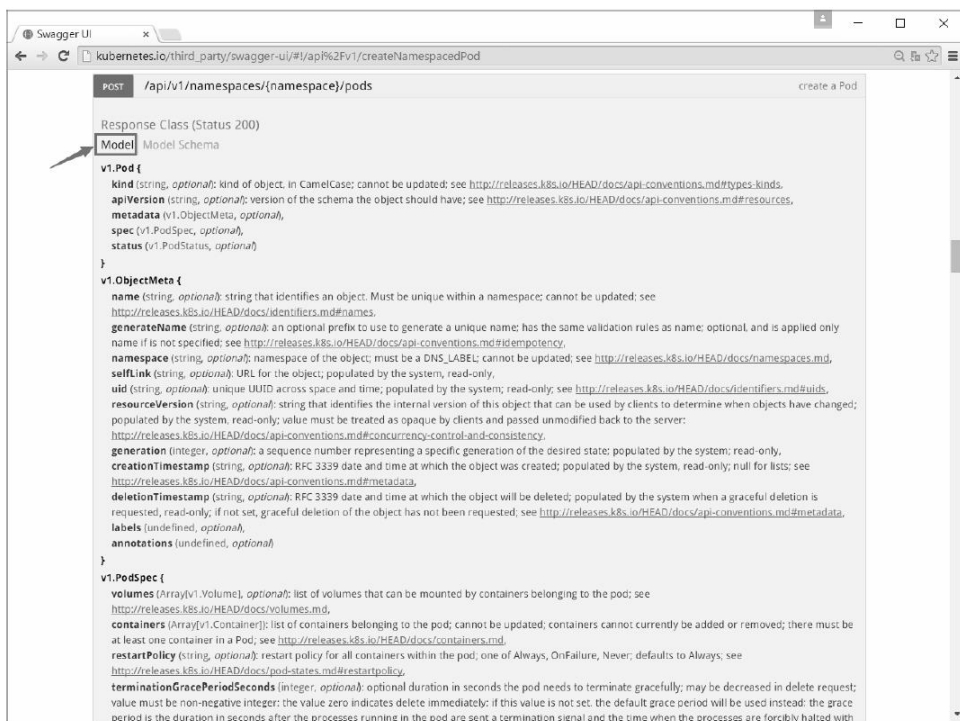


图4.5 Create a Pod API文本格式详细说明

我们看到，在Kubernetes API中，一个API的顶层（Top Level）元素由kind、apiVersion、metadata、spec和status等几个部分组成，接下来，我们分别对这几个部分进行说明。

kind表明对象有以下三大类别。

（1）对象（objects）：代表在系统中的一个永久资源（实体），例如Pod、RC、Service、Namespace及Node等。通过操作这些资源的属性，客户端可以对该对象进行创建、修改、删除和获取操作。

（2）列表（list）：一个或多个资源类别的集合。列表有一个通用元数据的有限集合。所有列表（lists）通过“items”域获得对象数组，例如PodLists、ServiceLists、NodeLists。大部分定义在系统中的

对象都有一个返回所有资源（**resource**）集合的端点，以及零到多个返回所有资源集合的子集的端点。某些对象有可能是单例对象（**singletons**），例如当前用户、系统默认用户等，这些对象没有列表。

（3）简单类别（**simple**）：该类别包含作用在对象上的特殊行为和非持久实体。该类别限制了使用范围，它有一个通用元数据的有限集合，例如**Binding**、**Status**。

apiVersion表明API的版本号，当前版本默认只支持**v1**。

Metadata是资源对象的元数据定义，是集合类的元素类型，包含一组由不同名称定义的属性。在**Kubernetes**中每个资源对象都必须包含以下3种**Metadata**。

（1）**namespace**：对象所属的命名空间，如果不指定，系统则会将对象置于名为“**default**”的系统命名空间中。

（2）**name**：对象的名字，在一个命名空间中名字应具备唯一性。

（3）**uid**：系统为每个对象生成的唯一ID，符合RFC 4122规范的定义。

此外，每种对象还应该包含以下几个重要元数据。

（1）**labels**：用户可定义的“标签”，键和值都为字符串的**map**，是对象进行组织和分类的一种手段，通常用于标签选择器（**Label Selector**），用来匹配目标对象。

(2) **annotations**: 用户可定义的“注解”，键和值都为字符串的map，被Kubernetes内部进程或者某些外部工具使用，用于存储和获取关于该对象的特定元数据。

(3) **resourceVersion**: 用于识别该资源内部版本号字符串，在用于Watch操作时，可以避免在GET操作和下一次Watch操作之间造成的信息不一致，客户端可以用它来判断资源是否改变。该值应该被客户端看作不透明，且不做任何修改就返回给服务端。客户端不应该假定版本信息具有跨命名空间、跨不同资源类别、跨不同服务器的含义。

(4) **creationTimestamp**: 系统记录创建对象时的时间戳，符合RFC 3339规范。

(5) **deletionTimestamp**: 系统记录删除对象时的时间戳，符合RFC 3339规范。

(6) **selfLink**: 通过API访问资源自身的URL，例如一个Pod的link可能是/api/v1/namespaces/default/pods/frontend-o8bg4。

spec是集合类的元素类型，用户对需要管理的对象进行详细描述的主体部分都在spec里给出，它会被Kubernetes持久化到etcd中保存，系统通过spec的描述来创建或更新对象，以达到用户期望的对象运行状态。spec的内容既包括用户提供的配置设置、默认值、属性的初始化值，也包括在对象创建过程中由其他相关组件（例如schedulers、auto-scalers）创建或修改的对象属性，比如Pod的Service IP地址。如果spec被删除，那么该对象将会从系统中被删除。

Status用于记录对象在系统中的当前状态信息，它也是集合类元素类型，**status**在一个自动处理的进程中被持久化，可以在流转的过程中生成。如果观察到一个资源丢失了它的状态（**Status**），则该丢失的状态可能被重新构造。以 **Pod** 为例，**Pod** 的 **status** 信息主要包括 **conditions**、**containerStatuses**、**hostIP**、**phase**、**podIP**、**startTime**等。其中比较重要的两个状态属性如下。

（1）**phase**：描述对象所处的生命周期阶段，**phase**的典型值是“**Pending**（创建中）”“**Running**”“**Active**（正在运行中）”或“**Terminated**（已终结）”，这几种状态对于不同的对象可能有轻微的差别，此外，关于当前**phase**附加的详细说明可能包含在其他域中。

（2）**condition**：表示条件，由条件类型和状态值组成，目前仅有一种条件类型**Ready**，对应的状态值可以为**True**、**False**或**Unknown**。一个对象可以具备多种**condition**，而**condition**的状态值也可能不断发生变化，**condition**可能附带一些信息，例如最后的探测时间或最后的转变时间。

4.2.2 API版本

为了在兼容旧版本的同时不断升级新的API，Kubernetes提供了多版本API的支持能力，每个版本的API通过一个版本号路径前缀进行区分，例如/api/v1beta3。通常情况下，新旧几个不同的API版本都能涵盖所有的Kubernetes资源对象，在不同的版本之间这些API接口存在一些细微差别。Kubernetes开发团队基于API级别选择版本而不是基于资源和域级别，是为了确保API能够描述一个清晰的连续的系统资源和行为的视图，能够控制访问的整个过程和控制实验性API的访问。

API及版本发布建议描述了版本升级的当前思路。版本v1beta1、v1beta2和v1beta3为不建议使用（Deprecated）的版本，请尽快转到v1版本。在2015年6月4日，Kubernetes v1版本API正式发布。版本v1beta1和v1beta2API在2015年6月1日被删除，版本v1beta3API在2015年7月6日被删除。

4.2.3 API详细说明

API资源使用REST模式，具体说明如下。

(1) **GET**/**<资源名的复数格式>**: 获得某一类型的资源列表，例如**GET/pods**返回一个Pod资源列表。

(2) **POST**/**<资源名的复数格式>**: 创建一个资源，该资源来自用户提供的JSON对象。

(3) **GET**/**<资源名复数格式>/<名字>**: 通过给出的名称(**Name**)获得单个资源，例如**GET/pods/first**返回一个名称为“first”的Pod。

(4) **DELETE**/**<资源名复数格式>/<名字>**: 通过给出的名字删除单个资源，在删除选项(**DeleteOptions**)中可以指定优雅删除(**Grace Deletion**)的时间(**GracePeriodSeconds**)，该可选项表明了从服务端接收到删除请求到资源被删除的时间间隔(单位为秒)。不同的类别(**Kind**)可能为优雅删除时间(**Grace Period**)申明默认值。用户提交的优雅删除时间将覆盖该默认值，包括值为0的优雅删除时间。

(5) **PUT**/**<资源名复数格式>/<名字>**: 通过给出的资源名和客户端提供的JSON对象来更新或创建资源。

(6) **PATCH**/**<资源名复数格式>/<名字>**: 选择修改资源详细指定的域。

对于 PATCH 操作，目前 Kubernetes API 通过相应的 HTTP 首部“Content-Type”对其进行识别。

目前支持以下三种类型的 PATCH 操作。

(1) JSON Patch, Content-Type: application/json-patch+json。在 RFC6902 的定义中，JSON Patch 是执行在资源对象上的一系列操作，例如 {“op”: “add”, “path”: “/a/b/c”, “value”: [“foo”, “bar”]}。详情请查看 RFC6902 说明，网址为 [HTTps: //tools.ietf.org/html/rfc6902](https://tools.ietf.org/html/rfc6902)。

(2) Merge Patch , Content-Type : application/merge-json-patch+json。在 RFC7386 的定义中，Merge Patch 必须包含对一个资源对象的部分描述，这个资源对象的部分描述就是一个 JSON 对象。该 JSON 对象被提交到服务端，并和服务端的当前对象合并，从而创建一个新的对象。详情请查看 RFC7386 说明，网址为 [HTTps: //tools.ietf.org/html/rfc7386](https://tools.ietf.org/html/rfc7386)。

(3) Strategic Merge Patch , Content-Type : application/strategic-merge-patch+json。Strategic Merge Patch 是一个定制化的 Merge Patch 实现。接下来将详细讲解 Strategic Merge Patch。

在标准的 JSON Merge Patch 中，JSON 对象总是被合并 (merge) 的，但是资源对象中的列表域总是被替换的。通常这不是用户所希望的。例如，我们通过下列定义创建一个 Pod 资源对象：

```
spec:
  containers:
    - name: nginx
      image: nginx-1.0
```

接着我们希望添加一个容器到这个Pod中，代码和上传的JSON对象如下所示：

```
PATCH /api/v1/namespaces/default/pods/pod-name
spec:
  containers:
    - name: log-tailer
      image: log-tailer-1.0
```

如果我们使用标准的Merge Patch，则其中的整个容器列表将被单个的“log-tailer”容器所替换。然而我们的目的是两个容器列表能够合并。

为了解决这个问题，Strategic Merge Patch通过添加元数据到API对象中，并通过这些新元数据来决定哪个列表被合并，哪个列表不被合并。当前这些元数据作为结构标签，对于API对象自身来说是合法的。对于客户端来说，这些元数据作为Swagger annotations也是合法的。在上述例子中，向“containers”中添加“patchStrategy”域，且它的值为“merge”，通过添加“patchMergeKey”，它的值为“name”。也就是说，“containers”中的列表将会被合并而不是替换，合并的依据为“name”域的值。

此外，Kubernetes API添加了资源变动的“观察者”模式的API接口。

- GET/watch/<资源名复数格式>：随时间变化，不断接收一连串的JSON对象，这些JSON对象记录了给定资源类别内所有资源对象

的变化情况。

- **GET/watch/资源名复数格式/<name>**: 随时间变化, 不断接收一连串的JSON对象, 这些JSON对象记录了某个给定资源对象的变化情况。

上述接口改变了返回数据的基本类别, **watch**动词返回的是一连串的JSON对象, 而不是单个的JSON对象。并不是所有的对象类别都支持“观察者”模式的API接口, 在后续的章节中将会说明哪些资源对象支持这种接口。

另外, Kubernetes还增加了HTTP Redirect与HTTP Proxy这两种特殊的API接口, 前者实现资源重定向访问, 后者则实现HTTP请求的代理。

4.2.4 API响应说明

API Server响应用户请求时附带一个状态码，该状态码符合HTTP规范。表4.1列出了API Server可能返回的状态码。

表4.1 API Server可能返回的状态码

状 态 码	编 码	描 述
200	OK	表明请求完全成功
201	Created	表明创建类的请求完全成功
204	NoContent	表明请求完全成功，同时 HTTP 响应不包含响应体。 在响应 OPTIONS 方法的 HTTP 请求时返回
307	TemporaryRedirect	表明请求资源的地址被改变，建议客户端使用 Location 首部给出的临时 URL 来定位资源
400	BadRequest	表明请求是非法的，建议客户不要重试，修改该请求
401	Unauthorized	表明请求能够到达服务端，且服务端能够理解用户请求，但是拒绝做更多的事情，因为客户端必须提供认证信息。如果客户端提供了认证信息，则返回该状态码，表明服务端指出所提供的认证信息不合适或非法
403	Forbidden	表明请求能够到达服务端，且服务端能够理解用户请求，但是拒绝做更多的事情，因为该请求被设置成拒绝访问。建议客户不要重试，修改该请求
404	NotFound	表明所请求的资源不存在。建议客户不要重试，修改该请求
405	MethodNotAllowed	表明请求中带有该资源不支持的方法。建议客户不要重试，修改该请求

续表

状 态 码	编 码	描 述
409	Conflict	表明客户端尝试创建的资源已经存在，或者由于冲突请求的更新操作不能被完成
422	UnprocessableEntity	表明由于所提供的作为请求部分的数据非法，创建或修改操作不能被完成
429	TooManyRequests	表明超出了客户端访问频率的限制或者服务端接收到多于它能处理的请求。建议客户端读取相应的 Retry-After 首部，然后等待该首部指出的时间后再重试
500	InternalServerError	表明服务端能被请求访问到，但是不能理解用户的请求；或者服务端内产生非预期中的一个错误，而且该错误无法被认知；或者服务端不能在一个合理的时间内完成处理（这可能由于服务器临时负载过重造成或者由于和其他服务器通信时的一个临时通信故障造成）
503	ServiceUnavailable	表明被请求的服务无效。建议客户不要重试修改该请求
504	ServerTimeout	表明请求在给定的时间内无法完成。客户端仅在为请求指定超时（Timeout）参数时会得到该响应

在调用API接口发生错误时，Kubernetes将会返回一个状态类别（Status Kind）。下面是两种常见的错误场景：

(1) 当一个操作不成功时（例如，当服务端返回一个非2xx HTTP状态码时）；

(2) 当一个HTTP DELETE方法调用失败时。

状态对象被编码成JSON格式，同时该JSON对象被作为请求的响应体。该状态对象包含人和机器使用的域，这些域中包含来自API的关于失败原因的详细信息。状态对象中的信息补充了对HTTP状态码的说明。例如：

```
$ curl -v -k -H "Authorization: Bearer
WhCDvq4VPpYhrcfmF6ei7V9qlbqTubUc"
HTTPS://10.240.122.184:443/api/v1/namespaces/default/pods/g
rafana
> GET /api/v1/namespaces/default/pods/grafana HTTP/1.1
> User-Agent: curl/7.26.0
> Host: 10.240.122.184
> Accept: */*
> Authorization: Bearer
WhCDvq4VPpYhrcfmF6ei7V9qlbqTubUc
>

< HTTP/1.1 404 Not Found
< Content-Type: application/json
< Date: Wed, 20 May 2015 18:10:42 GMT
< Content-Length: 232
<
```

```
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {},
  "status": "Failure",
  "message": "pods \"grafana\" not found",
  "reason": "NotFound",
  "details": {
    "name": "grafana",
    "kind": "pods"
  },
  "code": 404
}
```

- “status”域包含两个可能的值：Success和Failure。
- “message”域包含对错误的可读描述。
- “reason”域包含说明该操作失败原因的可读描述。如果该域的值为空，则表示该域内没有任何说明信息。“reason”域澄清HTTP状态码，但没有覆盖该状态码。
- “details”可能包含和“reason”域相关的扩展数据。每个“reason”域可以定义它的扩展的“details”域。该域是可选的，返回数据的格式是不确定的，不同的reason类型返回的“details”域的内容不一样。

4.3 使用Java程序访问Kubernetes API

本节介绍如何使用Java程序访问Kubernetes API。在Kubernetes的官网上列出了多个访问Kubernetes API的开源项目，其中有两个是用Java语言开发工具的开源项目，一个是OSGI，另一个是Fabric8。在本节所列的两个Java开发例子中，一个是基于Jersey的，另一个是基于Fabric8的。

4.3.1 Jersey

Jersey是一个RESTful请求服务JAVA框架。与Struts类似，它可以和Hibernate、Spring框架整合。通过它不仅方便开发RESTful Web Service，而且可以将它作为客户端方便地访问RESTful Web Service服务端。

如果没有一个好的工具包，则开发一个能够用不同的媒介（Media）类型无缝地暴露你的数据，以及很好地抽象客户、服务端通信的底层通信的RESTful Web Services，会很不容易。为了能够简化用Java开发RESTful Web Service及其客户端的流程，业界设计了JAX-RS API。Jersey RESTful Web Services框架是一个开源的高质量的框架，它为用JAVA语言开发RESTful Web Service及其客户端而生，支持JAX-RS APIs。Jersey不仅支持JAX-RS APIs，而且在此基础上扩展了API接口，这些扩展更加方便和简化了RESTful Web Services及其客户端的开发。

由于Kubernetes API Server是RESTful Web Service，因此此处选用Jersey框架开发RESTful Web Service客户端，用来访问Kubernetes API。在本例中选用的Jersey框架的版本为1.19，所涉及的Jar包如图4.6所示。
























 commons-codec-1.2.jar	2015/9/13 11:10	Executable Jar File	30 KB
 commons-httpclient-3.1.jar	2015/9/13 11:09	Executable Jar File	298 KB
 commons-logging-1.0.4.jar	2015/9/13 11:10	Executable Jar File	38 KB
 jackson-core-asl-1.9.2.jar	2015/2/11 5:41	Executable Jar File	223 KB
 jackson-jaxrs-1.9.2.jar	2015/2/11 5:41	Executable Jar File	18 KB
 jackson-mapper-asl-1.9.2.jar	2015/2/11 5:41	Executable Jar File	748 KB
 jackson-xc-1.9.2.jar	2015/2/11 5:41	Executable Jar File	27 KB
 jersey-apache-client-1.19.jar	2015/2/11 5:41	Executable Jar File	22 KB
 jersey-atom-abdera-1.19.jar	2015/2/11 5:41	Executable Jar File	20 KB
 jersey-client-1.19.jar	2015/2/11 5:41	Executable Jar File	131 KB
 jersey-core-1.19.jar	2015/2/11 5:41	Executable Jar File	427 KB
 jersey-guice-1.19.jar	2015/2/11 5:41	Executable Jar File	16 KB
 jersey-json-1.19.jar	2015/2/11 5:41	Executable Jar File	162 KB
 jersey-multipart-1.19.jar	2015/2/11 5:41	Executable Jar File	53 KB
 jersey-server-1.19.jar	2015/2/11 5:41	Executable Jar File	687 KB
 jersey-servlet-1.19.jar	2015/2/11 5:41	Executable Jar File	126 KB
 jersey-simple-server-1.19.jar	2015/2/11 5:41	Executable Jar File	12 KB
 jersey-spring-1.19.jar	2015/2/11 5:41	Executable Jar File	18 KB
 jettison-1.1.jar	2015/2/11 5:41	Executable Jar File	67 KB
 jsr311-api-1.1.1.jar	2015/2/11 5:41	Executable Jar File	46 KB
 oauth-client-1.19.jar	2015/2/11 5:41	Executable Jar File	15 KB
 oauth-server-1.19.jar	2015/2/11 5:41	Executable Jar File	30 KB
 oauth-signature-1.19.jar	2015/2/11 5:41	Executable Jar File	24 KB

图4.6 本例所涉及的Jar包

对Kubernetes API的访问包含如下三个方面。

(1) 指明访问资源的类型。

(2) 访问时的一些选项（参数），比如命名空间、对象的名称、过滤方式（标签和域）、子目录、访问的目标是否是代理和是否用watch方式访问等。

(3) 访问的方法，比如增、删、改、查。

在使用Jersey框架访问Kubernetes API之前，为这三个方面定义了三个对象。第1个定义的对象为ResourceType，它定义了访问资源的类型；第2个定义的对象是Params，它定义了访问API时的一些选项，以及通过这些选项如何生成完整的URI；第3个定义的对象是

RestfulClient，它是一个接口，该接口定义了访问API的方法（Method）。

ResourceType是一个ENUM类型的对象，定义了16种资源，代码如下：

```
package com.hp.k8s.apiclient.imp;

publicenum ResourceType {
    NODES("nodes"),
    NAMESPACES("namespaces"),
    SERVICES("services"),
    REPLICATIONCONTROLLERS("replicationcontrollers"),
    PODS("pods"),
    BINDINGS("bindings"),
    ENDPOINTS("endpoints"),
    SERVICEACCOUNTS("serviceaccounts"),
    SECRETS("secrets"),
    EVENTS("events"),
    COMPONENTSTATUSES("componentstatuses"),
    LIMITRANGES("limitranges"),
    RESOURCEQUOTAS("resourcequotas"),
    PODTEMPLATES("podtemplates"),
    PERSISTENTVOLUMECLAIMS("persistentvolumeclaims");
    PERSISTENTVOLUMES("persistentvolumes");
    private String type;
```

```

        private ResourceType(String type) {
            this.type = type;
        }

        public String getType() {
            return type;
        }
    }
}

```

Params对象的代码如下:

```

package com.hp.k8s.apiclient.imp;

import java.io.UnsupportedEncodingException;
import java.net.URLEncoder;
import java.util.List;
import java.util.Map;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Params {
    private static final Logger LOG =
LogManager.getLogger(Params.class.getName());
    private String namespace = null;
    private String name = null;
    private Map<String, String> fields = null;

```

```

private Map<String, String> labels = null;
private Map<String, String> notLabels = null;
private Map<String, List<String>> inLabels = null;
    private Map<String, List<String>> notInLabels =
null;

private String json = null;
private ResourceType resourceType = null;
private String subPath = null;
private boolean isVisitProxy = false;
private boolean isSetWatcher = false;

public String buildPath() {
    StringBuilder result = (isVisitProxy    new
StringBuilder("/proxy")
                                : (isSetWatcher    new
StringBuilder("/watch") : new StringBuilder(""))));
    if (null != namespace)

result.append("/namespaces/").append(namespace);

result.append("/").append(resourceType.getType());
    if (null != name)
        result.append("/").append(name);
    if (null != subPath)
        result.append("/").append(subPath);

```



```

        if (null != labels && !labels.isEmpty() ||
null != notLabels && !notLabels.isEmpty()
                                || null != inLabels &&
inLabels.size() > 0 || null != notInLabels &&
notInLabels.size() > 0
                                || null != fields &&
fields.size() > 0) {
                                StringBuilder labelSelectorStr =
null;
                                StringBuilder fieldSelectorStr =
null;
                                try {
                                                labelSelectorStr =
builderLabelSelector();
                                                fieldSelectorStr =
builderFiledSelector();
                                } catch
(UnsupportedEncodingException e1) {
                                                LOG.error(e1);
                                }

                                if (labelSelectorStr.length() +
fieldSelectorStr.length() > 0)
                                                result.append(" ");
                                if (labelSelectorStr.length() > 0)
{

```



```

        result.append(",");
    }

    result.append(URLEncoder.encode(key + "=",
labels.get(key), "GBK"));
    }
}

if (null != notLabels) {
    for (String key : labels.keySet())
{
        if (result.length() > 0) {
            result.append(",");
        }

result.append(URLEncoder.encode(key + "!="
labels.get(key),
"GBK"));
    }
}

if (null != inLabels) {
    for (String key :
inLabels.keySet()) {
        if (result.length() > 0) {

```

```

result.append(URLEncoder.encode(", ", "GBK"));
    }

result.append(URLEncoder.encode(key + " in (" +
listToString(inLabels.get(key), ",") + ")", "GBK"));
    }
}

    if (null != notInLabels) {
        for (String key :
inLabels.keySet()) {
            if (result.length() > 0) {

result.append(URLEncoder.encode(", ", "GBK"));
                }

result.append(URLEncoder.encode(key + " notin (" +
listToString(inLabels.get(key), ",") + ")", "GBK"));
            }
        }

    LOG.info("label result:" + result);
    return result;
}

```

```

private StringBuilder builderFiledSelector() throws

```

```

UnsupportedEncodingException {
    StringBuilder result = new StringBuilder();
    if (null != fields) {
        for (String key : fields.keySet())
        {
            if (result.length() > 0) {
                result.append(",");
            }

            result.append(URLEncoder.encode(key + "=" +
            fields.get(key), "GBK"));
        }
    }

    return result;
}

private String listToString(List<String> list,
String delim) {
    boolean isFirst = true;
    StringBuilder result = new StringBuilder();
    for (String str : list) {
        if (isFirst) {
            result.append(str);
            isFirst = false;
        } else {

```

```

        result.append(delim).append(str);
    }
}

    return result.toString();
}

public String getNamespace() {
    return namespace;
}

public void setNamespace(String namespace) {
    this.namespace = namespace;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public Map<String, String> getFields() {
    return fields;
}

```

```

    public void setFields(Map<String, String> fields) {
        this.fields = fields;
    }

    public Map<String, String> getLabels() {
        return labels;
    }

    public void setLabels(Map<String, String> labels) {
        this.labels = labels;
    }

    public String getJson() {
        return json;
    }

    public void setJson(String json) {
        this.json = json;
    }

    public ResourceType getResourceType() {
        return resourceType;
    }

    public void setResourceType(ResourceType
resourceType) {

```

```
        this.resourceType = resourceType;
    }

    public String getSubPath() {
        return subPath;
    }

    public void setSubPath(String subPath) {
        this.subPath = subPath;
    }

    public boolean isVisitProxy() {
        return isVisitProxy;
    }

    public void setVisitProxy(boolean isVisitProxy) {
        this.isVisitProxy = isVisitProxy;
    }

    public boolean isSetWatcher() {
        return isSetWatcher;
    }

    public void setSetWatcher(boolean isSetWatcher) {
        this.isSetWatcher = isSetWatcher;
    }
}
```



```
        public Map<String, String> getNotLabels() {
            return notLabels;
        }

        public void setNotLabels(Map<String, String>
notLabels) {
            this.notLabels = notLabels;
        }

        public Map<String, List<String>> getInLabels() {
            return inLabels;
        }

        public void setInLabels(Map<String, List<String>>
inLabels) {
            this.inLabels = inLabels;
        }

        public Map<String, List<String>> getNotInLabels() {
            return notInLabels;
        }

        public void setNotInLabels(Map<String, List<String>>
notInLabels) {
            this.notInLabels = notInLabels;
        }
    }
}
```

Params对象包含的属性说明如表4.2所示。

表4.2 Params对象包含的属性列表

属 性	说 明
namespace	String 类型属性，指明资源所在的命名空间，如果没有指定该值，则表明访问所有命名空间下的资源对象
name	String 类型属性，在访问单个资源对象时使用，如果没有指定该值，则表明访问该类资源列表
fields	Map<String, String>类型属性，通过资源对象的域值过滤访问结果
labels	Map<String, String>类型属性，通过指定的标签选择器列表来选择资源对象。选择出的资源对象包含标签列表中所列的标签（即 Map 的 key），且所选资源的标签的 value 和标签列表中的 value 值（即 Map 的 value）相等
notLabels	Map<String, String>类型属性，通过指定的标签选择器列表来选择资源对象。选择出的资源对象包含标签列表中所列的标签（即 Map 的 key），且所选资源的标签的 value 和标签列表中的 value 值（即 Map 的 value）不相等
inLabels	Map<String, List<String>>类型属性，通过指定的标签选择器列表来选择资源对象。Map 对象的 key 值为标签名称，Map 对象的 value 值为该标签可能包含的值
notInLabels	Map<String, List<String>>类型属性，通过指定的标签选择器列表来选择资源对象。Map 对象的 key 值为标签名称，Map 对象的 value 值为列表，表明资源对象包含和 key 值同名的标签，且这些标签的值不在该列表中
json	String 类型属性，在创建或修改资源对象时使用，用于向 API Server 提供资源对象的定义
resourceType	ResourceType 类型属性，用于指明访问资源对象的类型
subPath	String 类型属性，用于指明访问资源的子目录
isVisitProxy	Boolean 类型属性，用于指明是否通过 Proxy 的方式访问资源对象
isSetWatcher	Boolean 类型属性，表明是否通过 Watcher 方式访问资源对象

Params的buildPath方法用于构建访问URL的完整路径。

接口对象 RestfulClient 定义了访问 API 接口的所有方法（Method），其代码列表如下：

```
package com.hp.k8s.apiclient;

import com.hp.k8s.apiclient.imp.Params;

publicinterface RestfulClient {

    public String get(Params params); //获得单个资源对象
    public String list(Params params); //获得资源对象列表
    public String create(Params params); //创建资源对象
```

```

        public String delete(Params params); //删除某个资源对象
        public String update(Params params); //部分更新某个资源对象
        public String updateWithMediaType(Params params, String mediaType); //通过mediaType, 实现Merge
        public String replace(Params params); //替换某个资源对象
        public String options(Params params);
        public String head(Params params);
    }
}

```

其中get和list方法对应Kubernetes API的GET方法；create方法对应API中的POST方法；delete方法对应API中的DELETE方法；update方法对应API中的PATCH方法；replace方法对应API中的PUT方法；options方法对应API中的OPTIONS方法；head方法对应API中的HEAD方法。

该接口的基于Jersey框架的实现类如下所示：

```

package com.hp.k8s.apiclient.imp;

import javax.ws.rs.core.MediaType;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

```

```

import com.hp.k8s.apiclient.RestfulClient;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;

import
com.sun.jersey.api.client.config.DefaultClientConfig;

import
com.sun.jersey.client.urlconnection.URLConnectionClientHand
ler;

    publicclass JerseyRestfulClient implements
RestfulClient {

        privatestaticfinal Logger LOG =
LogManager.getLogger(RestfulClient.class.getName());

        privatestaticfinal String METHOD_PATCH = "PATCH";

        private String _baseUrl = null;
        Client _client = null;

        public JerseyRestfulClient(String baseUrl) {

            DefaultClientConfig config = new
DefaultClientConfig();

config.getProperties().put(URLConnectionClientHandler.PROPE
RTY_HTTP_URL_CONNECTION_SET_METHOD_WORKAROUND, true);

            _client = Client.create(config);

            this._baseUrl = baseUrl;

```

```

    }

    @Override
    public String get(Params params) {
        WebResource resource =
_client.resource(_baseUrl + params.buildPath());
        String response =
resource.accept(MediaType.APPLICATION_JSON_TYPE).
get(String.class);
        LOG.info("Get one resource:\n" + response);

        return response;
    }

    @Override
    public String list(Params params) {
        WebResource resource =
_client.resource(_baseUrl + params.buildPath());
        LOG.info("URL:" + _baseUrl +
params.buildPath());
        String response =
resource.accept(MediaType.APPLICATION_JSON_TYPE).
get(String.class);

        return response;
    }

```

```

        @Override
        public String create(Params params) {
            WebResource resource =
_client.resource(_baseUrl + params.buildPath());
            LOG.info("URL:" + _baseUrl +
params.buildPath());
            LOG.info("Create resource:" +
params.getJson());
            String response = (null ==
params.getJson())

resource.accept(MediaType.APPLICATION_JSON).post(String.class)

:
resource.type(MediaType.APPLICATION_JSON).accept(MediaType.
APPLICATION_JSON).post(String.class,

params.getJson());

            return response;
        }

```

```

        @Override
        public String delete(Params params) {
            WebResource resource =
_client.resource(_baseUrl + params.buildPath());
            String response =

```

```

resource.accept(MediaType.APPLICATION_JSON_TYPE).
delete(String.class);

        LOG.info("Delete resource " +
params.getResourceType().getType() + "/" + params.getName()
+ " result:\n"

        + response);

        return response;
    }

    @Override
    public String update(Params params) {
        return updateWithMediaType(params,
MediaType.APPLICATION_JSON);
    }

    @Override
    public String updateWithMediaType(Params params,
String mediaType) {

        WebResource resource =
_client.resource(_baseUrl + params.buildPath());
        LOG.info("URL:" + _baseUrl +
params.buildPath());

        LOG.info("Patch resource:" +
params.getJson());

        String response =
resource.type(mediaType).accept(MediaType.APPLICATION_

```

```
JSON_TYPE).method(METHOD_PATCH, String.class,
                    params.getJson());
        LOG.info("Update resource " +
params.buildPath() + " result:\n" + response);

        return response;
    }

    @Override
    public String replace(Params params) {
        WebResource resource =
_client.resource(_baseUrl + params.buildPath());
        LOG.info("URL:" + _baseUrl +
params.buildPath());
        LOG.info("Replace resource:" +
params.getJson());
        String response =
resource.type(MediaType.APPLICATION_JSON_TYPE).accept
(MediaType.APPLICATION_JSON_TYPE)
.put(String.class,
params.getJson());
        LOG.info("Replace resource " +
params.buildPath() + " result:\n" + response);

        return response;
    }
}
```



```

@Override
    public String options(Params params) {
        WebResource resource =
_client.resource(_baseUrl + params.buildPath());
        String response =
resource.type(MediaType.APPLICATION_JSON_TYPE).accept
(MediaType.TEXT_PLAIN_TYPE)
        .options(String.class);
        LOG.info("Get options for resource " +
params.getResourceType().getType() + "/" +
params.getName()
        + " result:\n" + response);

        return response;
    }

```

```

@Override
    public String head(Params params) {
        WebResource resource =
_client.resource(_baseUrl + params.buildPath());
        String response =
resource.accept(MediaType.TEXT_PLAIN_TYPE).head().
getResponseStatus().toString();
        LOG.info("Get head for resource " +
params.getResourceType().getType() + "/" + params.getName()
+ " result:\n"
        + response);
    }

```

```
        return response;
    }

    @Override
    public void close() {
        _client.destroy();
    }
}
```

该对象中包含如下代码:

```
config.getProperties().put(URLConnectionClientHandler.PROPE  
RTY_HTTP_URL_CONNECTION_SET_METHOD_WORKAROUND, true);
```

该段代码的作用是使Jersey客户端能够支持除标准REST方法外的方法，比如PATCH方法。该段代码能访问除watcher外的所有Kubernetes API接口，在后续的章节中我们会举例说明如何访问Kubernetes API。

4.3.2 Fabric8

Fabric8包含多款工具包，Kubernetes Client只是其中之一，也是Kubernetes官网中提到的Java Client API之一。本例子代码涉及的Jar包如图4.7所示。












 dnsjava-2.1.7.jar	2015/8/31 14:23	Executable Jar File	301 KB
 fabric8-utils-2.2.22.jar	2015/8/31 14:23	Executable Jar File	134 KB
 jackson-annotations-2.6.0.jar	2015/8/31 16:27	Executable Jar File	46 KB
 jackson-core-2.6.1.jar	2015/8/31 16:28	Executable Jar File	253 KB
 jackson-databind-2.6.1.jar	2015/8/31 15:56	Executable Jar File	1,140 KB
 jackson-dataformat-yaml-2.6.1.jar	2015/8/31 15:56	Executable Jar File	313 KB
 jackson-module-jaxb-annotations-2.6.0.jar	2015/8/31 16:24	Executable Jar File	32 KB
 json-20141113.jar	2015/8/31 14:23	Executable Jar File	64 KB
 kubernetes-api-2.2.22.jar	2015/8/31 14:22	Executable Jar File	72 KB
 kubernetes-client-1.3.8.jar	2015/8/31 15:37	Executable Jar File	2,262 KB
 kubernetes-model-1.0.12.jar	2015/8/31 15:56	Executable Jar File	2,308 KB
 log4j-api-2.3.jar	2015/8/31 16:18	Executable Jar File	133 KB
 log4j-core-2.3.jar	2015/8/31 15:56	Executable Jar File	808 KB
 log4j-slf4j-impl-2.3.jar	2015/8/31 15:56	Executable Jar File	23 KB
 oauth-20100527.jar	2015/8/31 15:56	Executable Jar File	44 KB
 openshift-client-1.3.2.jar	2015/8/31 14:23	Executable Jar File	24 KB
 slf4j-api-1.7.12.jar	2015/8/31 15:56	Executable Jar File	32 KB
 sundr-annotations-0.0.25.jar	2015/8/31 15:56	Executable Jar File	146 KB
 validation-api-1.1.0.Final.jar	2015/8/31 14:23	Executable Jar File	63 KB

图4.7 例子代码涉及的Jar包

因为该工具包已经对访问Kubernetes API客户端做了较好的封装，因此其访问代码比较简单，其具体的访问过程会在后续的章节举例说明。

Fabric 8的Kubernetes API客户端工具包只能访问Node、Service、Pod、Endpoints、Events、Namespace、PersistenetVolumeclaims、PersistenetVolume、ReplicationController、ResourceQuota、Secret和

ServiceAccount这几个资源类型，不能使用**OPTIONS**和**HEAD**方法访问资源，且不能以代理方式访问资源，但其对以**watcher**方式访问资源做了很好的支持。

4.3.3 使用说明

首先，举例说明对API资源的基本访问，也就是对资源的增、删、改、查，以及替换资源的status。其中会单独对Node和Pod的特殊接口做举例说明。表4.3列出了各资源对象的基本API接口。

表4.3 各资源对象的基本API接口

资源类型	方法	URL Path	说明	备注
NODES	GET	/api/v1/nodes	获取 Node 列表	
	POST	/api/v1/nodes	创建一个 Node 对象	
	DELETE	/api/v1/nodes/{name}	删除一个 Node 对象	
	GET	/api/v1/nodes/{name}	获取一个 Node 对象	
	PATCH	/api/v1/nodes/{name}	部分更新一个 Node 对象	
	PUT	/api/v1/nodes/{name}	替换一个 Node 对象	
NAMESPACES	GET	/api/v1/namespaces	获取 Namespace 列表	
	POST	/api/v1/namespaces	创建一个 Namespace 对象	
	DELETE	/api/v1/namespaces/{name}	删除一个 Namespace 对象	
	GET	/api/v1/namespaces/{name}	获取一个 Namespace 对象	
	PATCH	/api/v1/namespaces/{name}	部分更新一个 Namespace 对象	
	PUT	/api/v1/namespaces/{name}	替换一个 Namespace 对象	
	PUT	/api/v1/namespaces/{name}/finalize	替换一个 Namespace 对象的最终方案对象	在 Fabric8 中没有实现
	PUT	/api/v1/namespaces/{name}/status	替换一个 Namespace 对象的状态	在 Fabric8 中没有实现
SERVICES	GET	/api/v1/services	获取 Service 列表	
	POST	/api/v1/services	创建一个 Service 对象	
	GET	/api/v1/namespaces/{namespace}/services	获取某个 Namespace 下的 Service 列表	
	POST	/api/v1/namespaces/{namespace}/services	在某个 Namespace 下创建列表	
	DELETE	/api/v1/namespaces/{namespace}/services/{name}	删除某个 Namespace 下的一个 Service 对象	
	GET	/api/v1/namespaces/{namespace}/services/{name}	获取某个 Namespace 下的一个 Service 对象	
	PATCH	/api/v1/namespaces/{namespace}/services/{name}	部分更新某个 Namespace 下的一个 Service 对象	
	PUT	/api/v1/namespaces/{namespace}/services/{name}	替换某个 Namespace 下的一个 Service 对象	

续表

资源类型	方法	URL Path	说明	备注
REPLICATIONCONTROLLERS	GET	/api/v1/replicationcontrollers	获取 RC 列表	
	POST	/api/v1/replicationcontrollers	创建一个 RC 对象	
	GET	/api/v1/namespaces/{namespace}/replicationcontrollers	获取某个 Namespace 下的 RC 列表	
	POST	/api/v1/namespaces/{namespace}/replicationcontrollers	在某个 Namespace 下创建一个 RC 对象	
	DELETE	/api/v1/namespaces/{namespace}/replicationcontrollers/{name}	删除某个 Namespace 下的 RC 对象	
	GET	/api/v1/namespaces/{namespace}/replicationcontrollers/{name}	获取某个 Namespace 下的 RC 对象	
	PATCH	/api/v1/namespaces/{namespace}/replicationcontrollers/{name}	部分更新某个 Namespace 下的 RC 对象	
	PUT	/api/v1/namespaces/{namespace}/replicationcontrollers/{name}	替换某个 Namespace 下的 RC 对象	
PODS	GET	/api/v1/pods	获取一个 Pod 列表	
	POST	/api/v1/pods	创建一个 Pod 对象	
	GET	/api/v1/namespaces/{namespace}/pods	获取某个 Namespace 下的 Pod 列表	
	POST	/api/v1/namespaces/{namespace}/pods	在某个 Namespace 下创建一个 Pod 对象	
	DELETE	/api/v1/namespaces/{namespace}/pods/{name}	删除某个 Namespace 下的一个 Pod 对象	
	GET	/api/v1/namespaces/{namespace}/pods/{name}	获取某个 Namespace 下的一个 Pod 对象	
	PATCH	/api/v1/namespaces/{namespace}/pods/{name}	部分更新某个 Namespace 下的一个 Pod 对象	
	PUT	/api/v1/namespaces/{namespace}/pods/{name}	替换某个 Namespace 下的一个 Pod 对象	
	PUT	/api/v1/namespaces/{namespace}/pods/{name}/status	替换某个 Namespace 下的一个 Pod 对象状态	在 Fabric8 中没有实现
	POST	/api/v1/namespaces/{namespace}/pods/{name}/binding	创建某个 Namespace 下的一个 Pod 对象的 Binding	在 Fabric8 中没有实现
	GET	/api/v1/namespaces/{namespace}/pods/{name}/exec	连接到某个 Namespace 下的一个 Pod 对象，并执行 exec	在 Fabric8 中没有实现
	POST	/api/v1/namespaces/{namespace}/pods/{name}/exec	连接到某个 Namespace 下的一个 Pod 对象，并执行 exec	在 Fabric8 中没有实现

续表

资源类型	方法	URL Path	说明	备注
	GET	/api/v1/namespaces/{namespace}/pods/{name}/log	连接到某个 Namespace 下的一个 Pod 对象, 并获取 log 日志信息	在 Fabric8 中没有实现
	GET	/api/v1/namespaces/{namespace}/pods/{name}/portforward	连接到某个 Namespace 下的一个 Pod 对象, 并实现端口转发	在 Fabric8 中没有实现
	POST	/api/v1/namespaces/{namespace}/pods/{name}/portforward	连接到某个 Namespace 下的一个 Pod 对象, 并实现端口转发	在 Fabric8 中没有实现
BINDINGS	POST	/api/v1/bindings	创建一个 Binding 对象	
	POST	/api/v1/namespaces/{namespace}/bindings	在某个 Namespace 下创建一个 Binding 对象	
ENDPOINTS	GET	/api/v1/endpoints	获取 Endpoint 列表	
	POST	/api/v1/endpoints	创建一个 Endpoint 对象	
	GET	/api/v1/namespaces/{namespace}/endpoints	获取某个 Namespace 下的 Endpoint 对象列表	
	POST	/api/v1/namespaces/{namespace}/endpoints	在某个 Namespace 下创建一个 Endpoint 对象	
	DELETE	/api/v1/namespaces/{namespace}/endpoints/{name}	删除某个 Namespace 下的 Endpoint 对象	
	GET	/api/v1/namespaces/{namespace}/endpoints/{name}	获取某个 Namespace 下的 Endpoint 对象	
	PATCH	/api/v1/namespaces/{namespace}/endpoints/{name}	部分更新某个 Namespace 下的 Endpoint 对象	
	PUT	/api/v1/namespaces/{namespace}/endpoints/{name}	替换某个 Namespace 下的 Endpoint 对象	
SERVICEACCOUNTS	GET	/api/v1/serviceaccounts	获取 Serviceaccount 列表	
	POST	/api/v1/serviceaccounts	创建一个 Serviceaccount 对象	
	GET	/api/v1/namespaces/{namespace}/serviceaccounts	获取某个 Namespace 下的 Serviceaccount 对象列表	
	POST	/api/v1/namespaces/{namespace}/serviceaccounts	在某个 Namespace 下创建一个 Serviceaccount 对象	
	DELETE	/api/v1/namespaces/{namespace}/serviceaccounts/{name}	删除某个 Namespace 下的一个 Serviceaccount 对象	
	GET	/api/v1/namespaces/{namespace}/serviceaccounts/{name}	获取某个 Namespace 下的一个 Serviceaccount 对象	

续表

资源类型	方法	URL Path	说明	备注
	PATCH	/api/v1/namespaces/{namespace}/serviceaccounts/{name}	部分更新某个 Namespace 下的一个 Serviceaccount 对象	
	PUT	/api/v1/namespaces/{namespace}/serviceaccounts/{name}	替换某个 Namespace 下的一个 Serviceaccount 对象	
SECRETS	GET	/api/v1/secrets	获取 Secret 列表	
	POST	/api/v1/secrets	创建一个 Secret 对象	
	GET	/api/v1/namespaces/{namespace}/secrets	获取某个 Namespace 下的 Secret 列表	
	POST	/api/v1/namespaces/{namespace}/secrets	在某个 Namespace 下创建一个 Secret 对象	
	DELETE	/api/v1/namespaces/{namespace}/secrets/{name}	删除某个 Namespace 下的一个 Secret 对象	
	GET	/api/v1/namespaces/{namespace}/secrets/{name}	获取某个 Namespace 下的一个 Secret 对象	
	PATCH	/api/v1/namespaces/{namespace}/secrets/{name}	部分更新某个 Namespace 下的一个 Secret 对象	
	PUT	/api/v1/namespaces/{namespace}/secrets/{name}	替换某个 Namespace 下的一个 Secret 对象	
EVENTS	GET	/api/v1/events	获取 Event 列表	
	POST	/api/v1/events	创建一个 Event 对象	
	GET	/api/v1/namespaces/{namespace}/events	获取某个 Namespace 下的 Event 列表	
	POST	/api/v1/namespaces/{namespace}/events	在某个 Namespace 下创建一个 Event 对象	
	DELETE	/api/v1/namespaces/{namespace}/events/{name}	删除某个 Namespace 下的一个 Event 对象	
	GET	/api/v1/namespaces/{namespace}/events/{name}	获取某个 Namespace 下的一个 Event 对象	
	PATCH	/api/v1/namespaces/{namespace}/events/{name}	部分更新某个 Namespace 下的一个 Event 对象	
	PUT	/api/v1/namespaces/{namespace}/events/{name}	替换某个 Namespace 下的一个 Event 对象	
COMPONENTSTATUSES	GET	/api/v1/componentstatuses	获取 ComponentStatus 列表	
	GET	/api/v1/namespaces/{namespace}/componentstatuses	获取某个 Namespace 下的 Component Status 列表	

续表

资源类型	方法	URL Path	说明	备注
	GET	/api/v1/namespaces/{namespace}/componentstatuses/{name}	获取某个 Namespace 下的一个 ComponentStatus 对象	
LIMITRANGES	GET	/api/v1/limitranges	获取 LimitRange 列表	
	POST	/api/v1/limitranges	创建一个 LimitRange 对象	
	GET	/api/v1/namespaces/{namespace}/limitranges	获取某个 Namespace 下的 LimitRange 列表	
	POST	/api/v1/namespaces/{namespace}/limitranges	在某个 Namespace 下创建一个 LimitRange 对象	
	DELETE	/api/v1/namespaces/{namespace}/limitranges/{name}	删除某个 Namespace 下的一个 LimitRange 对象	
	GET	/api/v1/namespaces/{namespace}/limitranges/{name}	获取某个 Namespace 下的一个 LimitRange 对象	
	PATCH	/api/v1/namespaces/{namespace}/limitranges/{name}	部分更新某个 Namespace 下的一个 LimitRange 对象	
	PUT	/api/v1/namespaces/{namespace}/limitranges/{name}	替换某个 Namespace 下的一个 LimitRange 对象	
RESOURCEQUOTAS	GET	/api/v1/resourcequotas	获取 ResourceQuota 列表	
	POST	/api/v1/resourcequotas	创建一个 ResourceQuota 对象	
	GET	/api/v1/namespaces/{namespace}/resourcequotas	获取某个 Namespace 下的 ResourceQuota 列表	
	POST	/api/v1/namespaces/{namespace}/resourcequotas	在某个 Namespace 下创建一个 ResourceQuota 对象	
	DELETE	/api/v1/namespaces/{namespace}/resourcequotas/{name}	删除某个 Namespace 下的一个 ResourceQuota 对象	
	GET	/api/v1/namespaces/{namespace}/resourcequotas/{name}	获取某个 Namespace 下的一个 ResourceQuota 对象	
	PATCH	/api/v1/namespaces/{namespace}/resourcequotas/{name}	部分更新某个 Namespace 下的一个 ResourceQuota 对象	
	PUT	/api/v1/namespaces/{namespace}/resourcequotas/{name}	替换某个 Namespace 下的一个 ResourceQuota 对象	
	PUT	/api/v1/namespaces/{namespace}/resourcequotas/{name}/status	替换某个 Namespace 下的一个 ResourceQuota 对象状态	在 Fabric8 中没有实现

续表

资源类型	方法	URL Path	说明	备注
PODTEMPLATES	GET	/api/v1/podtemplates	获取 PodTemplate 列表	
	POST	/api/v1/podtemplates	创建一个 PodTemplate 对象	
	GET	/api/v1/namespaces/{namespace}/podtemplates	获取某个 Namespace 下的 PodTemplate 列表	
	POST	/api/v1/namespaces/{namespace}/podtemplates	在某个 Namespace 下创建一个 PodTemplate 对象	
	DELETE	/api/v1/namespaces/{namespace}/podtemplates/{name}	删除某个 Namespace 下的一个 PodTemplate 对象	
	GET	/api/v1/namespaces/{namespace}/podtemplates/{name}	获取某个 Namespace 下的一个 PodTemplate 对象	
	PATCH	/api/v1/namespaces/{namespace}/podtemplates/{name}	部分更新某个 Namespace 下的一个 PodTemplate 对象	
	PUT	/api/v1/namespaces/{namespace}/podtemplates/{name}	替换某个 Namespace 下的一个 PodTemplate 对象	
PERSISTENTVOLUMES	GET	/api/v1/persistentvolumes	获取 PersistentVolume 列表	
	POST	/api/v1/persistentvolumes	创建一个 PersistentVolume 对象	
	DELETE	/api/v1/persistentvolumes/{name}	删除一个 PersistentVolume 对象	
	GET	/api/v1/persistentvolumes/{name}	获取一个 PersistentVolume 对象	
	PATCH	/api/v1/persistentvolumes/{name}	部分更新一个 PersistentVolume 对象	
	PUT	/api/v1/persistentvolumes/{name}	替换一个 PersistentVolume 对象	
	PUT	/api/v1/persistentvolumes/{name}/status	替换一个 PersistentVolume 对象状态	在 Fabric8 中没有实现
PERSISTENTVOLUMECLAIMS	GET	/api/v1/persistentvolumeclaims	获取 PersistentVolumeClaim 列表	
	POST	/api/v1/persistentvolumeclaims	创建一个 PersistentVolumeClaim 对象	
	GET	/api/v1/namespaces/{namespace}/persistentvolumeclaims	获取某个 Namespace 下的 PersistentVolumeClaim 列表	
	POST	/api/v1/namespaces/{namespace}/persistentvolumeclaims	在某个 Namespace 下创建一个 PersistentVolumeClaim 对象	
	DELETE	/api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}	删除某个 Namespace 下的一个 PersistentVolumeClaim 对象	
	GET	/api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}	获取某个 Namespace 下的一个 PersistentVolumeClaim 对象	
	PATCH	/api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}	部分更新某个 Namespace 下的一个 PersistentVolumeClaim 对象	

续表

资源类型	方法	URL Path	说明	备注
	PUT	/api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}	替换某个 Namespace 下的一个 PersistentVolumeClaim 对象	
	PUT	/api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}/status	替换某个 Namespace 下的一个 PersistentVolumeClaim 对象状态	在 Fabric8 中没有实现

首先，举例说明如何通过API接口来创建资源对象。我们需要创建访问API Server的客户端，基于Jersey框架的代码如下：

```
RestfulClient _restfulClient = new  
JerseyRestfulClient("http://192.168.1.128:8080/api/v1");
```

其中，http://192.168.1.128: 8080 为 API Server 的地址。基于 Fabric8 框架的代码如下：

```
Config _conf = new Config();  
KubernetesClient_kube = new  
DefaultKubernetesClient("http://192.168.1.128: 8080");
```

分别通过上面的两个客户端创建 Namespace 资源对象，基于 Jersey 框架的代码如下：

```
private void testCreateNamespace() {  
    Params params = new Params();  
  
    params.setResourceType(ResourceType.NAMESPACES);  
  
    params.setJson(Utills.getJson("namespace.json"));  
  
    LOG.info("Result:" +  
_restfulClient.create(params));  
}
```

其中，“namespace.json”为创建 Namespace 资源对象的 JSON 定义，代码如下：

```
{
  "kind": "Namespace",
  "apiVersion": "v1",
  "metadata": {
    "name": "ns-sample"
  }
}
```

基于Fabric8框架的代码如下:

```
private void testCreateNamespace() {
    Namespace ns = new Namespace();
    ns.setApiVersion(ApiVersion.V_1);
    ns.setKind("Namespace");
    ObjectMeta om = new ObjectMeta();
    om.setName("ns-fabric8");
    ns.setMetadata(om);

    _kube.namespaces().create(ns);

    LOG.info(_kube.namespaces().list().getItems().size());
}
```

由于Fabric8框架对Kubernetes API对象做了很好的封装，对其中的大量对象都做了定义，所以用户可以通过其提供的资源对象去定义Kubernetes API对象，例如上面例子中的Namespace对象。Fabric8框架

中的kubernetes-model工具包用于API对象的封装。在上面的例子中，通过Fabric8框架提供的类创建了一个名为“ns-fabric8”的命名空间对象。

接下来我们会通过基于Jeysey框架的代码去创建两个Pod资源对象。在两个例子中，一个是在上面创建的“ns-sample”Namespace中创建Pod资源对象，另一个是为后续创建“cluster service”而创建的Pod资源对象。由于基于Fabric8框架创建Pod资源对象的方法很简单，因此不再用Fabric8框架对上述两个例子做说明。通过基于Jersey框架创建这两个Pod资源对象的代码如下：

```
private void testCreatePod() {
    Params params = new Params();
    params.setResourceType(ResourceType.PODS);

    params.setJson(Utils.getJson("podInNs.json"));
    params.setNamespace("ns-sample");
    LOG.info("Result:" +
        _restfulClient.create(params));

    params.setJson(Utils.getJson("pod4ClusterService.json"));
    LOG.info("Result:" +
        _restfulClient.create(params));
}
```

其中，podInNs.json和pod4ClusterService.json是创建两个Pod资源对象的定义。podInNs.json文件的内容如下：

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "pod-sample-in-namespace",
    "namespace": "ns-sample"
  },
  "spec": {
    "containers": [{
      "name": "mycontainer",
      "image": "kubeguide/redis-master"
    }]
  }
}
```

pod4ClusterService.json文件的内容如下：

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "pod-sample-4-cluster-service",
    "namespace": "ns-sample",
    "labels": {
```

```

        "k8s-cs": "kube-cluster-service",
        "k8s-test": "kube-cluster-test",
        "k8s-sample-app": "kube-service-sample",
        "kkk": "bbb"
    }
},
"spec":{
    "containers":[{
        "name":"mycontainer",
        "image":"kubeguide/redis-master"
    }]
}
}

```

下面的例子代码用于获取Pod资源列表，其中第1部分代码用于获取所有的Pod资源对象，第2、3部分代码主要是列举如何使用标签选择Pod资源对象，最后一部分代码用于举例说明如何使用field选择Pod资源对象。代码如下：

```

private void testGetPodList() {
    Params params = new Params();
    params.setResourceType(ResourceType.PODS);
    LOG.info("Result:" +
_restfulClient.list(params));

    Map<String, String> labels = new
HashMap<String, String>();

```

```

        labels.put("k8s-cs", "kube-cluster-
service");

        labels.put("k8s-sample-app", "kube-service-
sample");

        params.setLabels(labels);

        LOG.info("Result:" +
_restfulClient.list(params));

        params.setLabels(null);

        Map<String, List<String>> inLabels = new
HashMap<String, List<String>>();
        List list = new ArrayList<String>();
        list.add("kube-cluster-service");
        list.add("kube-cluster");
        inLabels.put("k8s-cs", list);
        params.setInLabels(inLabels);

        LOG.info("Result:" +
_restfulClient.list(params));

        params.setInLabels(null);

        Map<String, String> fields = new
HashMap<String, String>();
        fields.put("metadata.name", "pod-sample-4-
cluster-service");

        params.setNamespace("ns-sample");
        params.setFields(fields);

        LOG.info("Result:" +

```



```
_restfulClient.list(params));  
    }
```

接下来的例子代码用于替换一个Pod对象，在通过Kubernetes API替换一个Pod资源对象时需要注意两点：

（1）在替换该资源对象前，先从API中获取该资源对象的JSON对象，然后在该JSON对象的基础上修改需要替换的部分；

（2）在Kubernetes API提供的接口中，PUT方法（replace）只支持替换容器的image部分。

代码如下：

```
private void testReplacePod() {  
    Params params = new Params();  
    params.setNamespace("ns-sample");  
    params.setName("pod-sample-in-namespace");  
  
    params.setJson(Utils.getJson("pod4Replace.json"));  
    params.setResourceType(ResourceType.PODS);  
  
    LOG.info("Result:" +  
_restfulClient.replace(params));  
}
```

其中，pod4Replace.json的内容如下：

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "pod-sample-in-namespace",
    "namespace": "ns-sample",
    "selfLink": "/api/v1/namespaces/ns-sample/pods/pod-
sample-in-namespace",
    "uid": "084ff63e-59d3-11e5-8035-000c2921ba71",
    "resourceVersion": "45450",
    "creationTimestamp": "2015-09-13T04:51:01Z"
  },
  "spec": {
    "volumes": [
      {
        "name": "default-token-szoje",
        "secret": {
          "secretName": "default-token-szoje"
        }
      }
    ],
    "containers": [
      {
        "name": "mycontainer",
        "image": "centos",
        "resources": {},
```

```
"volumeMounts": [  
    {  
      "name": "default-token-szoje",  
      "readOnly": true,  
      "mountPath":  
"/var/run/secrets/kubernetes.io/serviceaccount"  
    }  
  ],  
  "terminationMessagePath": "/dev/termination-log",  
  "imagePullPolicy": "IfNotPresent"  
}  
],  
"restartPolicy": "Always",  
"dnsPolicy": "ClusterFirst",  
"serviceAccountName": "default",  
"serviceAccount": "default",  
"nodeName": "192.168.1.129"  
},  
"status": {  
  "phase": "Running",  
  "conditions": [  
    {  
      "type": "Ready",  
      "status": "True"  
    }  
  ],  
  "hostIP": "192.168.1.129",
```

```

"podIP": "10.1.10.66",
"startTime": "2015-09-11T15:17:28Z",
"containerStatuses": [
    {
      "name": "mycontainer",
      "state": {
        "running": {
          "startedAt": "2015-09-11T15:17:30Z"
        }
      },
      "lastState": {},
      "ready": true,
      "restartCount": 0,
      "image": "kubeguide/redis-master",
      "imageID":
        "docker://5630952871a38cddffda9ec611f5978ab0933628fcd54cd7d
        7677ce6b17de33f",
      "containerID":
        "docker://7bf0d454c367418348711556e667fd1ef6a04d7153d
        24bfcac2e2e06da634a9f"
    }
  ]
}

```

接下来的两个例子实现了4.2.4节中提到的两种Merge方式：Merge Patch和Strategic Merge Patch。

第1种Merge方式的示例如下:

```
private void testUpdatePod1() {
    Params params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");

    params.setJson(Utils.getJson("pod4MergeJsonPatch.json"));
    params.setResourceType(ResourceType.PODS);

    LOG.info("Result:" +
        _restfulClient.updateWithMediaType(params, "application/
        merge-patch+json"));
}
```

其中, pod4MergeJsonPatch.json的内容如下:

```
{
  "metadata":{
    "labels":{
      "k8s-cs": "kube-cluster-service",
      "k8s-test": "kube-cluster-test",
      "k8s-sa555mple-app": "kube-service-sample",
      "kkk": "bbb4444"
    }
  }
}
```

第2种Merge方式（Strategic Merge Patch）的示例如下：

```
private void testUpdatePod2() {
    Params params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");

    params.setJson(Utils.getJson("pod4StrategicMerge.json"));
    params.setResourceType(ResourceType.PODS);

    LOG.info("Result:" +
        _restfulClient.updateWithMediaType(params,
            "application/strategic-merge-patch+json"));
}
```

其中， pod4StrategicMerge.json的内容如下：

```
{
  "spec": {
    "containers": [{
      "name": "mycontainer",
      "image": "centos",
      "patchStrategy": "merge",
      "patchMergeKey": "name"
    }]
  }
}
```

接下来实现了修改Pod资源对象的状态，代码如下：

```
private void testStatusPod() {  
    Params params = new Params();  
    params.setNamespace("ns-sample");  
    params.setName("pod-sample-in-namespace");  
    params.setSubPath("/status");  
  
    params.setJson(Utils.getJson("pod4Status.json"));  
    params.setResourceType(ResourceType.PODS);  
  
    _restfulClient.replace(params);  
}
```

其中， pod4Status.json的内容如下：

```
{  
  "kind": "Pod",  
  "apiVersion": "v1",  
  "metadata": {  
    "name": "pod-sample-in-namespace",  
    "namespace": "ns-sample",  
    "selfLink": "/api/v1/namespaces/ns-sample/pods/pod-sample-in-namespace",  
    "uid": "ad1d803f-59ec-11e5-8035-000c2921ba71",  
    "resourceVersion": "51640",  
    "creationTimestamp": "2015-09-13T07:54:35Z"
```

```

    },
    "spec": {
      "volumes": [
        {
          "name": "default-token-szoje",
          "secret": {
            "secretName": "default-token-szoje"
          }
        }
      ],
      "containers": [
        {
          "name": "mycontainer",
          "image": "kubeguide/redis-master",
          "resources": {},
          "volumeMounts": [
            {
              "name": "default-token-szoje",
              "readOnly": true,
              "mountPath":
"/var/run/secrets/kubernetes.io/serviceaccount"
            }
          ],
          "terminationMessagePath": "/dev/termination-log",
          "imagePullPolicy": "IfNotPresent"
        }
      ],

```



```
"restartPolicy": "Always",
"dnsPolicy": "ClusterFirst",
"serviceAccountName": "default",
"serviceAccount": "default",
"nodeName": "192.168.1.129"
  },
"status": {
  "phase": "Unknown",
  "conditions": [
    {
      "type": "Ready",
      "status": "false"
    }
  ],
  "hostIP": "192.168.1.129",
  "podIP": "10.1.10.79",
  "startTime": "2015-09-11T18:21:02Z",
  "containerStatuses": [
    {
      "name": "mycontainer",
      "state": {
        "running": {
          "startedAt": "2015-09-11T18:21:03Z"
        }
      }
    }
  ],
  "lastState": {},
  "ready": true,
```

```
        "restartCount": 0,
        "image": "kubeguide/redis-master",
                                                    "imageID":
"docker://5630952871a38cddffda9ec611f5978ab0933628fcd54cd
7d7677ce6b17de33f",
                                                    "containerID":
"docker://b0e2312643e9a4b59cf1ff5fb7a8468c5777180d5a
8ea5f2f0c9dfddcf3f4cd2"
    }
]
}
}
```

接下来实现了查看Pod的log日志功能，代码如下：

```
private void testLogPod() {
    Params params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");
    params.setSubPath("/log");
    params.setResourceType(ResourceType.PODS);

    _restfulClient.get(params);
}
```

下面通过API访问Node的多种接口，代码如下：

```
private void testPoxyNode() {
    Params params = new Params();
    params.setName("192.168.1.129");
    params.setSubPath("pods");
    params.setVisitProxy(true);
    params.setResourceType(ResourceType.NODES);
    _restfulClient.get(params);

    params = new Params();
    params.setName("192.168.1.129");
    params.setSubPath("stats");
    params.setVisitProxy(true);
    params.setResourceType(ResourceType.NODES);
    _restfulClient.get(params);

    params = new Params();
    params.setName("192.168.1.129");
    params.setSubPath("spec");
    params.setVisitProxy(true);
    params.setResourceType(ResourceType.NODES);
    _restfulClient.get(params);

    params = new Params();
    params.setName("192.168.1.129");
    params.setSubPath("run/ns-sample/pod/pod-sample-in-namespace");
}
```

```
        params.setVisitProxy(true);
        params.setResourceType(ResourceType.NODES);
        _restfulClient.get(params);

        params = new Params();
        params.setName("192.168.1.129");
        params.setSubPath("metrics");
        params.setVisitProxy(true);
        params.setResourceType(ResourceType.NODES);
        _restfulClient.get(params);
    }
```

最后，举例说明如何通过API删除资源对象pod，代码如下：

```
private void testDeletePod() {
    Params params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");
    params.setResourceType(ResourceType.PODS);
    LOG.info("Result:" +
        _restfulClient.delete(params));
}
```

通过API接口除了能够对资源对象实现前面列出的基本操作外，还涉及两类特殊接口，一类是WATCH，一类是PROXY。这两类特殊接口所包含的接口如表4.4所示。

表4.4 两类特殊接口所包含的接口

资源类型	类别	方法	URL Path	说明
NODES	WATCH	GET	/api/v1/watch/nodes	监听所有节点的变化
		GET	/api/v1/watch/nodes/{name}	监听单个节点的变化
	PROXY	DELETE	/api/v1/proxy/nodes/{name}/{path:~}	代理 DELETE 请求到节点的某个子目录
		GET	/api/v1/proxy/nodes/{name}/{path:~}	代理 GET 请求到节点的某个子目录
		HEAD	/api/v1/proxy/nodes/{name}/{path:~}	代理 HEAD 请求到节点的某个子目录
		OPTIONS	/api/v1/proxy/nodes/{name}/{path:~}	代理 OPTIONS 请求到节点的某个子目录
		POST	/api/v1/proxy/nodes/{name}/{path:~}	代理 POST 请求到节点的某个子目录
		PUT	/api/v1/proxy/nodes/{name}/{path:~}	代理 PUT 请求到节点的某个子目录
		DELETE	/api/v1/proxy/nodes/{name}	代理 DELETE 请求到节点
		GET	/api/v1/proxy/nodes/{name}	代理 GET 请求到节点
		HEAD	/api/v1/proxy/nodes/{name}	代理 HEAD 请求到节点
		OPTIONS	/api/v1/proxy/nodes/{name}	代理 OPTIONS 请求到节点
		POST	/api/v1/proxy/nodes/{name}	代理 POST 请求到节点
		PUT	/api/v1/proxy/nodes/{name}	代理 PUT 请求到节点
SERVICES	WATCH	GET	/api/v1/watch/services	监听所有 Service 的变化
		GET	/api/v1/watch/namespaces/{namespace}/services	监听某个 Namespace 下所有 Service 的变化

续表

资源类型	类别	方法	URL Path	说明
		GET	/api/v1/watch/namespaces/{namespace}/services/{name}	监听某个 Service 的变化
	PROXY	DELETE	/api/v1/proxy/namespaces/{namespace}/services/{name}/{path:~}	代理 DELETE 请求到 Service 的某个子目录
		GET	/api/v1/proxy/namespaces/{namespace}/services/{name}/{path:~}	代理 GET 请求到 Service 的某个子目录
		HEAD	/api/v1/proxy/namespaces/{namespace}/services/{name}/{path:~}	代理 HEAD 请求到 Service 的某个子目录
		OPTIONS	/api/v1/proxy/namespaces/{namespace}/services/{name}/{path:~}	代理 OPTIONS 请求到 Service 的某个子目录
		POST	/api/v1/proxy/namespaces/{namespace}/services/{name}/{path:~}	代理 POST 请求到 Service 的某个子目录
		PUT	/api/v1/proxy/namespaces/{namespace}/services/{name}/{path:~}	代理 PUT 请求到 Service 的某个子目录
		DELETE	/api/v1/proxy/namespaces/{namespace}/services/{name}	代理 DELETE 请求到 Service
		GET	/api/v1/proxy/namespaces/{namespace}/services/{name}	代理 GET 请求到 Service
		HEAD	/api/v1/proxy/namespaces/{namespace}/services/{name}	代理 HEAD 请求到 Service
		OPTIONS	/api/v1/proxy/namespaces/{namespace}/services/{name}	代理 OPTIONS 请求到 Service
		POST	/api/v1/proxy/namespaces/{namespace}/services/{name}	代理 POST 请求到 Service
		PUT	/api/v1/proxy/namespaces/{namespace}/services/{name}	代理 PUT 请求到 Service
REPLICATIONCONTROLLER	WATCH	GET	/api/v1/watch/replicationcontrollers	监听所有 RC 的变化
		GET	/api/v1/watch/namespaces/{namespace}/replicationcontrollers	监听某个 Namespace 下所有 RC 的变化
		GET	/api/v1/watch/namespaces/{namespace}/replicationcontrollers/{name}	监听某个 RC 的变化
PODS	WATCH	GET	/api/v1/watch/pods	监听所有 Pod 的变化
		GET	/api/v1/watch/namespaces/{namespace}/pods	监听某个 Namespace 下所有 Pod 的变化

续表

资源类型	类别	方法	URL Path	说明
	PROXY	GET	/api/v1/watch/namespaces/{namespace}/pods/{name}	监听某个 Pod 的变化
		DELETE	/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:.*}	代理 DELETE 请求到 Pod 的某个子目录
		GET	/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:.*}	代理 GET 请求到 Pod 的某个子目录
		HEAD	/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:.*}	代理 HEAD 请求到 Pod 的某个子目录
		OPTIONS	/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:.*}	代理 OPTIONS 请求到 Pod 的某个子目录
		POST	/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:.*}	代理 POST 请求到 Pod 的某个子目录
		PUT	/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:.*}	代理 PUT 请求到 Pod 的某个子目录
		DELETE	/api/v1/namespaces/{namespace}/pods/{name}/proxy	代理 DELETE 请求到 Pod
		GET	/api/v1/namespaces/{namespace}/pods/{name}/proxy	代理 GET 请求到 Pod
		HEAD	/api/v1/namespaces/{namespace}/pods/{name}/proxy	代理 HEAD 请求到 Pod
		OPTIONS	/api/v1/namespaces/{namespace}/pods/{name}/proxy	代理 OPTIONS 请求到 Pod
		POST	/api/v1/namespaces/{namespace}/pods/{name}/proxy	代理 POST 请求到 Pod
		PUT	/api/v1/namespaces/{namespace}/pods/{name}/proxy	代理 PUT 请求到 Pod
		DELETE	/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:.*}	代理 DELETE 请求到 Pod 的某个子目录
		GET	/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:.*}	代理 GET 请求到 Pod 的某个子目录
		HEAD	/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:.*}	代理 HEAD 请求到 Pod 的某个子目录
		OPTIONS	/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:.*}	代理 OPTIONS 请求到 Pod 的某个子目录
		POST	/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:.*}	代理 POST 请求到 Pod 的某个子目录

续表

资源类型	类别	方法	URL Path	说明
		PUT	/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:~}	代理 PUT 请求到 Pod 的某个子目录
		DELETE	/api/v1/proxy/namespaces/{namespace}/pods/{name}	代理 DELETE 请求到 Pod
		GET	/api/v1/proxy/namespaces/{namespace}/pods/{name}	代理 GET 请求到 Pod
		HEAD	/api/v1/proxy/namespaces/{namespace}/pods/{name}	代理 HEAD 请求到 Pod
		OPTIONS	/api/v1/proxy/namespaces/{namespace}/pods/{name}	代理 OPTIONS 请求到 Pod
		POST	/api/v1/proxy/namespaces/{namespace}/pods/{name}	代理 POST 请求到 Pod
		PUT	/api/v1/proxy/namespaces/{namespace}/pods/{name}	代理 PUT 请求到 Pod
ENDPOINTS	WATCH	GET	/api/v1/watch/endpoints	监听所有 Endpoint 的变化
		GET	/api/v1/watch/namespaces/{namespace}/endpoints	监听某个 Namespace 下所有 Endpoint 的变化
		GET	/api/v1/watch/namespaces/{namespace}/endpoints/{name}	监听某个 Endpoint 的变化
SERVICEACCOUNT	WATCH	GET	/api/v1/watch/serviceaccounts	监听所有 ServiceAccount 的变化
		GET	/api/v1/watch/namespaces/{namespace}/serviceaccounts	监听某个 Namespace 下所有 ServiceAccount 的变化
		GET	/api/v1/watch/namespaces/{namespace}/serviceaccounts/{name}	监听某个 ServiceAccount 的变化
SECRET	WATCH	GET	/api/v1/watch/secrets	监听所有 Secret 的变化
		GET	/api/v1/watch/namespaces/{namespace}/secrets	监听某个 Namespace 下所有 Secret 的变化
		GET	/api/v1/watch/namespaces/{namespace}/secrets/{name}	监听某个 Secret 的变化
EVENTS	WATCH	GET	/api/v1/watch/events	监听所有 Event 的变化
		GET	/api/v1/watch/namespaces/{namespace}/events	监听某个 Namespace 下所有 Event 的变化

续表

资源类型	类别	方法	URL Path	说明
		GET	/api/v1/watch/namespaces/{namespace}/events/{name}	监听某个 Event 的变化
LIMITRANGES	WATCH	GET	/api/v1/watch/limitranges	监听所有 Event 的变化
		GET	/api/v1/watch/namespaces/{namespace}/limitranges	监听某个 Namespace 下所有 Event 的变化
		GET	/api/v1/watch/namespaces/{namespace}/limitranges/{name}	监听某个 Event 的变化
RESOURCEQUOTAS	WATCH	GET	/api/v1/watch/resourcequotas	监听所有 ResourceQuota 的变化
		GET	/api/v1/watch/namespaces/{namespace}/resourcequotas	监听某个 Namespace 下所有 ResourceQuota 的变化
		GET	/api/v1/watch/namespaces/{namespace}/resourcequotas/{name}	监听某个 ResourceQuota 的变化
PODTEMPLATES	WATCH	GET	/api/v1/watch/podtemplates	监听所有 PodTemplate 的变化
		GET	/api/v1/watch/namespaces/{namespace}/podtemplates	监听某个 Namespace 下所有 PodTemplate 的变化
		GET	/api/v1/watch/namespaces/{namespace}/podtemplates/{name}	监听某个 PodTemplate 的变化
PERSISTENTVOLUMES	WATCH	GET	/api/v1/watch/persistentvolumes	监听所有 PersistentVolume 的变化
		GET	/api/v1/watch/persistentvolumes/{name}	监听某个 PersistentVolume 的变化
PERSISTENTVOLUMECLAIMS	WATCH	GET	/api/v1/watch/persistentvolumeclaims	监听所有 PersistentVolumeClaim 的变化
		GET	/api/v1/watch/namespaces/{namespace}/persistentvolumeclaims	监听某个 Namespace 下所有 PersistentVolumeClaim 的变化
		GET	/api/v1/watch/namespaces/{namespace}/persistentvolumeclaims/{name}	监听某个 PersistentVolumeClaim 的变化

下面基于Fabric8实现对资源对象的监听（Watch），代码如下：

```

private void testWatcher() {
    _kube.pods().watch(new
io.fabric8.kubernetes.client.Watcher<Pod>() {
        @Override
        public void eventReceived(Action action, Pod pod) {
            System.out.println(action + ": " +
pod);
        }
    });
}

```

```
        @Override
        public void onClose(KubernetesClientException e) {
            System.out.println("Closed: " + e);
        }
    });
}
```

接下来基于Jersey框架实现通过Proxy方式访问Pod。由于API Server针对Pod资源提供了两种Proxy访问接口，所以下面分别用两段代码进行示例说明。代码如下：

```
private void testPoxyPod() {
    //访问第1种proxy接口
    Params params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");
    params.setSubPath("/proxy");
    params.setResourceType(ResourceType.PODS);

    _restfulClient.get(params);

    //访问第2种proxy接口
    params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");
    params.setVisitProxy(true);
}
```

```
params.setResourceType(ResourceType.PODS);
```

```
_restfulClient.get(params);
```

```
}
```

第5章 Kubernetes运维指南

为了让容器应用在Kubernetes集群中运行得更加有效，对Kubernetes集群本身也需要进行相应的配置和管理。本章将从Kubernetes集群管理、高级案例及Trouble Shooting等方面对Kubernetes集群的运维和查错进行详细说明，最后对Kubernetes1.3版本开发中的新功能进行介绍。

5.1 Kubernetes集群管理指南

本节将从Node的管理、Label的管理、Namespace资源共享、资源配额管理、集群Master高可用及集群监控等方面，对Kubernetes集群本身的运维管理进行详细说明。

5.1.1 Node的管理

1.Node的隔离与恢复

在硬件升级、硬件维护等情况下，我们需要将某些Node进行隔离，脱离Kubernetes集群的调度范围。Kubernetes提供了一种机制，既可以将Node纳入调度范围，也可以将Node脱离调度范围。

创建配置文件 `unschedule_node.yaml`，在 `spec` 部分指定 `unschedulable` 为 `true`:

```
apiVersion: v1
kind: Node
metadata:
  name: k8s-node-1
  labels:
    kubernetes.io/hostname: k8s-node-1
spec:
  unschedulable: true
```

然后，通过 `kubectl replace` 命令完成对Node状态的修改:

```
$ kubectl replace -f unschedule_node.yaml
node "k8s-node-1" replaced
```

查看Node的状态，可以观察到在Node的状态中增加了一项SchedulingDisabled:

# kubectl get nodes		
NAME	STATUS	AGE
k8s-node-1	Ready,SchedulingDisabled	1h

对于后续创建的Pod，系统将不会再向该Node进行调度。

也可以不使用配置文件，直接使用kubectl patch命令完成:

```
$ kubectl patch node k8s-node-1 -p '{"spec":
{"unschedulable":true}}'
```

需要注意的是，将某个Node脱离调度范围时，在其上运行的Pod并不会自动停止，管理员需要手动停止在该Node上运行的Pod。

同样，如果需要将某个Node重新纳入集群调度范围，则将unschedulable设置为false，再次执行kubectl replace或kubectl patch命令就能恢复系统对该Node的调度。

在Kubernetes当前的版本中，kubectl的子命令cordon和uncordon也用于实现将Node进行隔离和恢复调度的操作。

例如，使用kubectl cordon<node_name>对某个Node进行隔离调度操作:

```
# kubectl cordon k8s-node-1
node "k8s-node-1" cordoned
```

```
# kubectl get nodes
```

NAME	STATUS	AGE
k8s-node-1	Ready,SchedulingDisabled	1h

使用 `kubectl uncordon`

使用 `kubectl uncordon<node_name>` 对某个Node进行恢复调度操作:

```
# kubectl uncordon k8s-node-1
```

```
node "k8s-node-1" uncordoned
```

```
# kubectl get nodes
```

NAME	STATUS	AGE
k8s-node-1	Ready	1h

2.Node的扩容

在实际生产系统中会经常遇到服务器容量不足的情况，这时就需要购买新的服务器，然后将应用系统进行水平扩展来完成对系统的扩容。

在Kubernetes集群中，一个新Node的加入是非常简单的。在新的Node节点上安装Docker、kubelet和kube-proxy服务，然后配置kubelet和kube-proxy的启动参数，将Master URL指定为当前Kubernetes集群Master的地址，最后启动这些服务。通过kubelet默认的自动注册机制，新的Node将会自动加入现有的Kubernetes集群中，如图5.1所示。

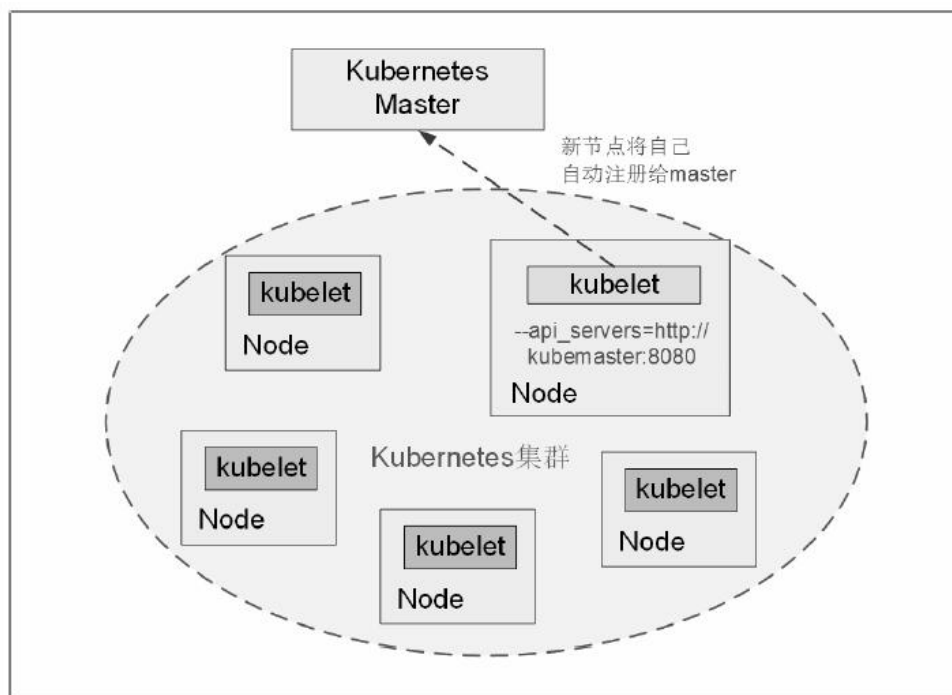


图5.1 新节点自动注册完成扩容

Kubernetes Master在接受了新**Node**的注册之后，会自动将其纳入当前集群的调度范围内，在之后创建容器时，就可以向新的**Node**进行调度了。

通过这种机制，**Kubernetes**实现了集群中**Node**的扩容。

5.1.2 更新资源对象的Label

Label（标签）作为用户可灵活定义的对象属性，在正在运行的资源对象上，仍然可以随时通过`kubectl label`命令对其进行增加、修改、删除等操作。

例如，我们要给已创建的Pod“redis-master-bobr0”添加一个标签`role=backend`:

```
$ kubectl label pod redis-master-bobr0 role=backend
pod "redis-master-bobr0" labeled
```

查看该Pod的Label:

```
$ kubectl get pods -Lrole
```

	NAME	READY	STATUS	RESTARTS
AGE	ROLE			
	redis-master-bobr0	1/1	Running	0
	backend			3m

删除一个Label时，只需在命令行最后指定Label的key名并与一个减号相连即可:

```
$ kubectl label pod redis-master-bobr0 role-
pod "redis-master-bobr0" labeled
```

修改一个Label的值时，需要加上--overwrite参数:

```
$ kubectl label pod redis-master-bobr0 role=master --  
overwrite  
pod "redis-master-bobr0" labeled
```

5.1.3 Namespace: 集群环境共享与隔离

在一个组织内部，不同的工作组可以在同一个Kubernetes集群中工作，Kubernetes通过命名空间和Context的设置来对不同的工作组进行区分，使得它们既可以共享同一个Kubernetes集群的服务，也能够互不干扰，如图5.2所示。

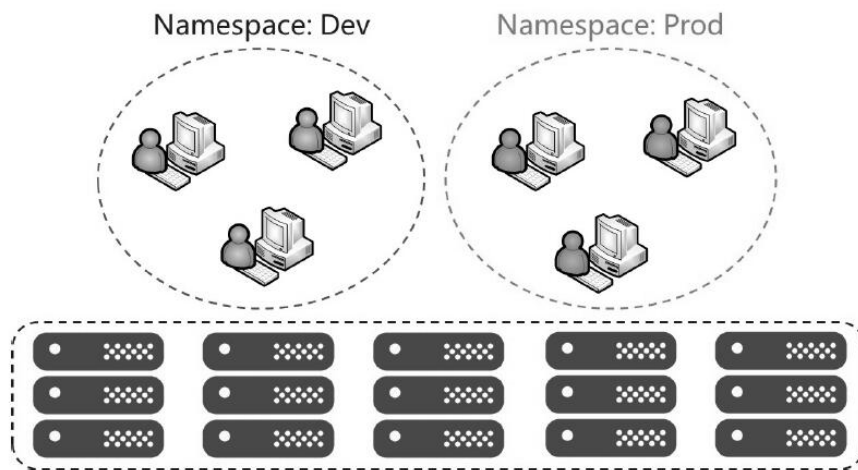


图5.2 集群环境共享和隔离

假设在我们的组织中有两个工作组：开发组和生产运维组。开发组在Kubernetes集群中需要不断创建、修改、删除各种Pod、RC、Service等资源对象，以便实现敏捷开发的过程。而生产运维组则需要使用严格的权限设置来确保生产系统中的Pod、RC、Service处于正常运行状态且不会被误操作。

1. 创建namespace

为了在Kubernetes集群中实现这两个分组，首先需要创建两个命名空间。

namespace-development.yaml:

```
apiVersion: v1
kind: Namespace
metadata:
  name: development
```

namespace-production.yaml:

```
apiVersion: v1
kind: Namespace
metadata:
  name: production
```

使用kubectl create命令完成命名空间的创建:

```
$ kubectl create -f namespace-development.yaml
namespaces/development
```

```
$ kubectl create -f namespace-production.yaml
namespaces/production
```

查看系统中的命名空间:

```
$ kubectl get namespaces
```

NAME	LABELS	STATUS
default	<none>	Active
development	name=development	Active
production	name=production	Active

2.定义Context（运行环境）

接下来，需要为这两个工作组分别定义一个Context，即运行环境。这个运行环境将属于某个特定的命名空间。

通过kubectl config set-context命令定义Context，并将Context置于之前创建的命名空间中：

```
$ kubectl config set-cluster kubernetes-cluster --
server=https://192.168.1.128:8080
```

```
$ kubectl config set-context ctx-dev --
namespace=development --cluster=kubernetes-cluster --
user=dev
```

```
$ kubectl config set-context ctx-prod --
namespace=production --cluster=kubernetes-cluster --
user=prod
```

使用kubectl config view命令查看已定义的Context：

```
$ kubectl config view
```

```
apiVersion: v1
```

```
clusters:
- cluster:
    server: http://192.168.1.128:8080
    name: kubernetes-cluster
contexts:
- context:
    cluster: kubernetes-cluster
    namespace: development
    name: ctx-dev
- context:
    cluster: kubernetes-cluster
    namespace: production
    name: ctx-prod
current-context: ctx-dev
kind: Config
preferences: {}
users: []
```

注意，通过`kubectl config`命令在`${HOME}/.kube`目录下生成了一个名为`config`的文件，文件内容即以`kubectl config view`命令查看到的内容。所以，也可以通过手工编辑该文件的方式来设置Context。

3. 设置工作组在特定Context环境中工作

使用`kubectl config use-context<context_name>`命令来设置当前的运行环境。

下面的命令将把当前运行环境设置为“ctx-dev”:

```
$ kubectl config use-context ctx-dev
```

通过这个命令，当前的运行环境即被设置为开发组所需的环境。之后的所有操作都将在名为“development”的命名空间中完成。

现在，以redis-slave RC为例创建两个Pod:

redis-slave-controller.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-slave
  labels:
    name: redis-slave
spec:
  replicas: 2
  selector:
    name: redis-slave
  template:
    metadata:
      labels:
        name: redis-slave
    spec:
      containers:
        - name: slave
```



```
image: kubeguide/guestbook-redis-slave
ports:
- containerPort: 6379
```

```
$ kubectl create -f redis-slave-controller.yaml
replicationcontrollers/redis-slave
```

查看创建好的Pod:

```
$ kubectl get pods
```

	NAME		READY	STATUS
RESTARTS	AGE			
	redis-slave-0feq9	1/1	Running	0
6m				
	redis-slave-6i0g4	1/1	Running	0
6m				

可以看到容器被正确创建并运行起来了。而且，由于当前的运行环境是`ctx-dev`，所以不会影响到生产运维组的工作。

让我们切换到生产运维组的运行环境:

```
$ kubectl config use-context ctx-prod
```

查看RC和Pod:

```
$ kubectl get rc
```

	CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR
--	------------	--------------	----------	----------

REPLICAS

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

结果为空，说明看不到开发组创建的RC和Pod。

现在我们为生产运维组也创建两个redis-slave的Pod:

```
$ kubectl create -f redis-slave-controller.yaml  
replicationcontrollers/redis-slave
```

查看创建好的Pod:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
redis-slave-a4m7s	1/1	Running	0	12s
redis-slave-xyrkk	1/1	Running	0	12s

可以看到容器被正确创建并运行起来了，并且当前的运行环境是ctx-prod，也不会影响开发组的工作。

至此，我们为两个工作组分别设置了两个运行环境，在设置好当前的运行环境时，各工作组之间的工作将不会相互干扰，并且它们都能够在同一个Kubernetes集群中同时工作。

5.1.4 Kubernetes资源管理

本章从计算资源管理（**Compute Resources**）、资源配置范围管理（**LimitRange**）、服务质量管理（**QoS**）及资源配额管理（**ResourceQuota**）等方面，对Kubernetes集群内的资源管理进行详细说明，结合实践操作、常见问题分析和一个完整的示例，对Kubernetes集群资源管理相关的运维工作提供指导。

1. 计算资源管理（**Compute Resources**）

在配置Pod的时候，我们可以为其中的每个容器指定需要使用的计算资源（CPU和内存）。

计算资源的配置项分为两种：一种是资源请求（**Resource Requests**，简称**Requests**），表示容器希望被分配到的、可完全保证的资源量，**Requests**的值会提供给Kubernetes调度器（**Kubernetes Scheduler**）以便于优化基于资源请求的容器调度；另外一种资源限制（**Resource Limits**，简称**Limits**），**Limits**是容器最多能使用到的资源量的上限，这个上限值会影响节点上发生资源竞争时的解决策略。

当前版本的Kubernetes中，计算资源的资源类型分为两种：**CPU**和**内存（Memory）**。这两种资源类型都有一个基本单位：对于**CPU**而言，基本单位是核心数（**Cores**）；而内存的基本单位是字节数（**Bytes**）。**CPU**和**内存**一起构成了目前Kubernetes中的计算资源（也可简称为资源）。

计算资源是可计量的，能被申请、分配和使用的基础资源，这使之区别于API资源（API Resources，例如Pod和services等）。

1) Pod和容器的Requests和Limits

Pod中的每个容器都可以配置以下4个参数。

- `spec.container[].resources.requests.cpu`。
- `spec.container[].resources.limits.cpu`。
- `spec.container[].resources.requests.memory`。
- `spec.container[].resources.limits.memory`。

这四个参数分别对应容器的CPU和内存的Requests和Limits，它们具有以下特点。

- Requests和Limits都是可选的。在某些集群中如果在Pod创建或者更新的时候，没设置资源限制或者资源请求值，那么可能会使用系统提供一个默认值，这个默认值取决于集群的配置。
- 如果Request没有配置，那么默认会被设置为等于Limits。
- 而任何情况下Limits都应该设置为大于或者等于Requests。

以CPU为例，图5.3显示了未设置CPU Limits和设置CPU Limits的CPU使用率的区别。

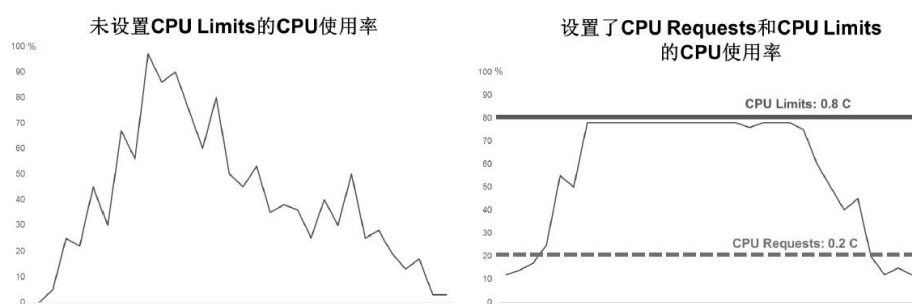


图5.3 未设置和设置了CPU Limits的CPU使用率样例

尽管Requests和Limits只能设置到容器上，但是设置Pod级别的Requests和Limits能极大程度上提高我们对Pod管理的便利性和灵活性，因此Kubernetes中提供对Pod级别的Requests和Limits配置。对于CPU和内存而言，Pod的Requests或Limits是指该Pod中所有容器的Requests或Limits的总和（Pod中没设置Request或Limits的容器，该项的值被当作0或者按照集群配置的默认值来计算）。下面对CPU和内存这两种计算资源各自的特点进行说明。

（1）CPU

CPU的Requests和Limits是通过CPU数（cpus）来度量的。CPU资源值支持最多三位小数：如果一个容器的spec.container[].resources.requests.cpu设置为0.5，那么它会获得半个CPU；同理如果设置为1，就会获得1个CPU。0.1CPU等价于100m CPU（100millicpu），而在Kubernetes API中自动将这种小数0.1转化为100m，因此CPU的小数最多支持三位数字，而Kubernetes官方也更推荐直接使用形如100m的millicpu作为计量单位。

CPU资源值是绝对值，而不是相对值：比如0.1CPU不管是在单核或者多核机器上都是一样的，都严格等于0.1CPU core。

（2）内存（Memory）

内存的Requests和Limits计量单位是字节数（Bytes）。内存值使用整数或者定点整数加上国际单位制（International System of Units）来表示。国际单位制包括十进制的E、P、T、G、M、K、m，或二进制的Ei、Pi、Ti、Gi、Mi、Ki。比如：KiB与MiB是二进制表示的字节

单位，而常见的KB与MB则是十进制表示的字节单位。两种方式的区别举例说明如下：

1KB (kilobyte) = 1000bytes = 8000bits

1KiB (kibibyte) = 2^{10} bytes = 1024bytes = 8192bits

因此，下面几种内存配置的意思是一样的：128974848、129e6、129M、123Mi

Kubernetes的计算资源单位是大小写敏感的，因为m可以表示千分之一单位（milli unit），而M可以表示十进制的1000，两者的含义不同；同理可知，小写的k不是一个合法的资源单位。

以一个Pod中的资源配置为例：

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
    - name: db
      image: mysql
      resources:
        requests:
          memory: "64Mi"
          cpu: "250m"
        limits:
```

```
        memory: "128Mi"
        cpu: "500m"
-   name: wp
    image: wordpress
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

该Pod包含两个容器，每个容器配置的Requests都是0.25CPU和64MiB（ 2^{26} bytes）内存，而配置的Limits都是0.5CPU和128MiB（ 2^{27} bytes）内存。

这个Pod的Requests和Limits等于Pod中所有容器对应配置的总和，所以Pod的Requests是0.5CPU和128MiB（ 2^{27} bytes）内存，Limits是1CPU和256MiB（ 2^{28} bytes）内存。

2) 基于Requests和Limits的Pod调度机制

当一个Pod创建成功时，Kubernetes调度器（Scheduler）为该Pod选择一个节点（Node）来执行。对于每种计算资源（CPU和内存）而言，每个节点都有一个能用于运行Pod的最大容量值。调度器在调度时，首先要确保调度后该节点上所有Pod的CPU和内存的Requests总和不能超过该节点能提供给Pod使用的CPU和内存的最大容量值。

例如某个节点上CPU资源充足，而内存为4GB，其中3GB可以运行Pod，某Pod的内存Requests为1GB、Limits为2GB，那么这个节点上最多可运行3个这种Pod。

这里需要注意的是：可能某些节点上的实际资源使用量非常低，但是如果该节点上已运行Pod配置的Requests值的总和已经非常高，再加上需要调度的Pod的Requests值会直接超过该节点提供给Pod的资源容量上限，Kubernetes仍然不会将Pod调度到这个节点上。这是因为如果Kubernetes将Pod调度到该节点上，那么如果后面该节点上运行的Pod面临服务峰值等情况，可能会导致Pod资源短缺的情况发生。

接着上面的例子，假设该节点已经启动3个Pod实例，而这3个Pod的实际内存使用都不足500MB，那么理论上该节点的可用内存应该大于1.5GB，但是由于该节点的Pod Requests总和已经等于节点的可用内存上限，因此Kubernetes不会再将任何Pod实例调度到该节点上执行。

3) Requests和Limits资源配置机制

当kubelet启动Pod的一个容器时，它会将容器的Requests和Limits值转化为相应的容器启动参数传递给容器执行器（Docker或者是rkt）。

如果容器的执行环境是Docker，那么容器的4个参数是这样传递给Docker的。

(1) spec.container[].resources.requests.cpu

这个参数会转化为core数（比如配置的100m会转化为0.1），然后乘以1024，再将这个结果作为--cpu-shares参数的值传递给docker run命

令。在`docker run`命令中，`--cpu-share`参数是一个相对权重值（Relative Weight），这个相对权重值会决定Docker在资源竞争时分配给容器的资源比例。举例说明`--cpu-shares`参数在Docker中的含义：比如两个容器的CPU Requests分别设置为1和2，那么容器在`docker run`启动时对应的`--cpu-shares`参数值分别为1024和2048，在主机CPU资源产生竞争时，Docker会尝试按照1:2的配比将CPU资源分配给这两个容器使用。

这里需要区分清楚的是：这个参数对于Kubernetes而言是绝对值，主要用于Kubernetes调度和管理的依据（参见下文QoS章节）；同时这个参数值会设置为`--cpu-shares`参数传递给Docker，`--cpu-shares`参数对于Docker而言又是相对值，主要用于资源分配比例。这两种用途的作用范围不同，所以并不会发生冲突。

(2) `spec.container[].resources.limits.cpu`

这个参数会转化为millicore数（比如配置的1会转化为1000，而配置的100m转化为100），将此值乘以100000，再除以1000，然后将结果值作为`--cpu-quota`参数的值传递给`docker run`命令。`docker run`命令中另外一个参数`--cpu-period`默认设置为100000，表示Docker重新计量和分配CPU的使用时间间隔为100000微秒（100毫秒）。

Docker的`--cpu-quota`参数和`--cpu-period`参数一起配合完成对容器CPU的使用限制：比如Kubernetes中配置容器的CPU Limits为0.1，那么计算后`--cpu-quota`为10000，而`--cpu-period`为100000，这意味着Docker在100毫秒内最多给该容器分配10毫秒*core的计算资源用量， $10/100=0.1\text{core}$ 的结果与Kubernetes配置的意义是一致的。

注意：如果 kubelet 启动参数 `--cpu-cfs-quota` 设置为 `true`，那么 kubelet 会强制要求所有 Pod 都必须配置 CPU Limits（如果 Pod 没配置，而集群提供了默认配置也可以）。而从 Kubernetes 1.2 版本开始，这个 `--cpu-cfs-quota` 启动参数的默认值就是 `true`。

(3) `spec.container[].resources.requests.memory`

这个参数值只提供给 Kubernetes 调度器（Kubernetes Scheduler）作为调度和管理的依据，不会作为任何参数传递给 Docker。

(4) `spec.container[].resources.limits.memory`

这个参数值会转化为单位为 bytes 的整数，数值会作为 `--memory` 参数传递给 `docker run` 命令。

如果一个容器在运行过程中使用了超出了其内存 Limits 配置的内存限制值，那么它可能会被“杀掉”，如果这个容器是一个可重启的容器，那么之后它会被 kubelet 重新启动起来。因此容器的 Limits 配置需要进行准确的测试和评估。

与内存 Limits 不同的是 CPU 在容器技术中属于可压缩资源，因此对于 CPU 的 Limits 配置一般不会引发因偶然超标使用而导致容器被系统“杀掉”的情况。

4) 计算资源使用情况监控

Pod 的资源用量会作为 Pod 的状态信息一同上报给 Master。如果集群中配置了 Heapster 来监控集群的性能数据，那么还可以从 Heapster 中查看 Pod 的资源用量信息。

5) 计算资源相关常见问题分析

(1) Pod状态为pending，错误信息为FailedScheduling。

如果Kubernetes调度器（Kubernetes Scheduler）在集群中找不到合适的节点来运行Pod，那么这个Pod会一直处于未调度状态，直到调度器找到合适的节点为止。每次调度器尝试调度失败，Kubernetes都会产生一个事件（event），我们可以通过下面这种方式来查看事件的信息：

```
$ kubectl describe pod frontend | grep -A 3 Events
```

Events:

	FirstSeen	LastSeen	Count	From
Subobject	PathReason			Message
	36s	5s	6	
{scheduler }		FailedScheduling	Failed for reason	
		PodExceedsFreeCPU and possibly others		

在上面这个例子中，名为frontend的Pod由于节点的CPU资源不足而调度失败（PodExceedsFreeCPU），同样，如果内存不足也可能导致调度失败（PodExceedsFreeMemory）。

如果一个或者多个Pod调度失败且有这类错误，那么我们可以尝试以下几种解决方法。

- 添加更多的节点到集群中。
- 停止一些不必要的运行中的Pod，释放资源。

- 检查Pod的配置，错误的配置可能导致该Pod永远都无法被调度执行。比如如果整个集群中所有节点都只有1CPU，而Pod配置的CPU Requests为2，那么该Pod就不会被调度执行。

我们可以使用`kubectl describe nodes`命令来查看集群中节点的计算资源容量和已使用量：

```
$ kubectl describe nodes k8s-node-1
```

```
Name:                                k8s-node-1
```

```
...
```

```
Capacity:
```

```
  cpu:                               1
```

```
  memory:    464Mi
```

```
  pods:                               40
```

```
Allocated resources (total requests):
```

```
  cpu:                               910m
```

```
  memory:    2370Mi
```

```
  pods:                               4
```

```
...
```

```
Pods:                                (4 in total)
```

Namespace	Name
CPU(milliCPU)	Memory(bytes)
frontend	webserver-ffj8j
500 (50% of total)	2097152000 (50% of total)
kube-system	fluentd-cloud-
logging-k8s-node-1	100 (10% of total)
209715200 (5% of total)	

kube-system	kube-dns-v8-qopgw
310 (31% of total)	178257920 (4% of total)
TotalResourceLimits:	
CPU(millicpu):	910 (91% of total)
Memory(bytes):	2485125120 (59% of total)
...	

超过可用资源容量上限（Capacity）和已分配资源量（Allocated resources）差额的Pod无法运行在该Node上。这个例子中，如果一个Pod的Requests超过90millicpus或者超过1341MiB内存，那么就无法运行在这个节点上。

在后面的资源配额（Resource Quota）章节中，我们还可以配置针对一组Pod的Requests和Limits总量的限制，这种限制可以作用于命名空间，通过这种方式我们可以防止一个命名空间下的用户将所有资源全部据为己有。

（2）容器被强行终止（Terminated）

如果容器使用的资源超过了它配置的Limits，那么该容器可能会被强制终止。我们可以通过kubect describe pod命令来确认容器是否是因为这个原因被终止的：

```
$ kubectl describe pod simmemleak-hra99
```

Name:	simmemleak-hra99
Namespace:	default
Image(s):	saadali/simmemleak
Node:	192.168.18.3

Labels: name=simmemleak
Status: Running
Reason:
Message:
IP: 172.17.1.3
Replication Controllers: simmemleak (1/1
replicas created)
Containers:
simmemleak:
Image: saadali/simmemleak
Limits:
cpu: 100m
memory: 50Mi
State: Running
Started: Tue, 07 Jul 2015
12:54:41 -0700
Last Termination State: Terminated
Exit Code: 1
Started: Fri, 07 Jul 2015
12:54:30 -0700
Finished: Fri, 07 Jul 2015
12:54:33 -0700
Ready: False
Restart Count: 5
Conditions:
Type Status
Ready False

Events:

	FirstSeen		LastSeen
Count	From		SubobjectPath
Reason	Message		
	Tue, 07 Jul 2015 12:53:51 -0700	Tue, 07 Jul 2015	
12:53:51	-0700 1	{scheduler }	
scheduled	Successfully assigned simmemleak-hra99 to		
kubernetes-node-tf0f			
	Tue, 07 Jul 2015 12:53:51 -0700	Tue, 07 Jul 2015	
12:53:51	-0700 1 {kubelet kubernetes-node-tf0f}		
implicitly required container POD	pulled	Pod	
container image "gcr.io/google_containers/pause:0.8.0"			
already present on machine			
	Tue, 07 Jul 2015 12:53:51 -0700	Tue, 07 Jul 2015	
12:53:51	-0700 1 {kubelet kubernetes-node-tf0f}		
implicitly required container POD	created	Created	
with docker id 6a41280f516d			
	Tue, 07 Jul 2015 12:53:51 -0700	Tue, 07 Jul 2015	
12:53:51	-0700 1 {kubelet kubernetes-node-tf0f}		
implicitly required container POD	started	Started	
with docker id 6a41280f516d			
	Tue, 07 Jul 2015 12:53:51 -0700	Tue, 07 Jul 2015	
12:53:51	-0700 1 {kubelet kubernetes-node-tf0f}		
spec.containers{simmemleak}	created	Created	
with docker id 87348f12526a			

Restart Count: 5说明这个名为simmemleak的容器被强制终止并重启了5次。

我们可以在使用kubect1 get pod命令时添加-o go-template=...格式参数来读取已终止容器之前的状态信息:

```
$ kubect1 get pod -o go-
template='{{range.status.containerStatuses}}{{"Container
Name:      "}}{{.name}}{{"\r\nLastState:      "}}{{.lastState}}
{{end}}}' simmemleak-60xbc
Container Name: simmemleak
LastState: map[terminated:map[exitCode:137 reason:OOM
Killed startedAt:2015-07-07T20:58:43Z finishedAt:2015-07-
07T20:58:43Z
containerID:docker://0e4095bba1feccdfef9fb6ebffe972b4b142
85d5acdec6f0d3ae8a22fad8b2]]
```

这里我们可以看到这个容器因为reason: OOM Killed而被强制终止,说明这个容器的内存超过了限制 (Out of Memory)。

6) 计算资源管理的演进

当前版本的Kubernetes中的Requests和Limits都是作用于容器级别的,未来Kubernetes计划增加对直接作用于Pod级别的资源配置的支持,这种资源配置是能被Pod内的所有容器共享的,包括emptyDir这种Pod级别的Volume。

从资源的种类来看，目前Kubernetes只能支持CPU和内存两种计算资源类型，在后续的版本中，Kubernetes计划支持更多的资源类型，包括节点磁盘空间资源，还将支持自定义的资源类型。

2.资源的配置范围管理 (LimitRange)

默认情况下，Kubernetes的Pod会以无限制的CPU和内存运行。这也就意味着Kubernetes系统中任何的Pod都可以使用其所在节点上的所有可用的CPU和内存。通过配置Pod的计算资源Requests和Limits，我们可以限制Pod的资源使用，但对于Kubernetes集群管理员而言，配置每一个Pod的Requests和Limits是烦琐且限制性过强的。更多的时候，我们需要的是对集群内Request和Limits的配置做一个全局的统一的限制。常见的配置场景如下。

- 集群中的每个节点有2GB内存，集群管理员不希望任何Pod申请超过2GB的内存：因为整个集群中没有任何节点能满足超过2GB内存的请求。如果某个Pod的内存配置超过2GB，那么该Pod将永远都无法被调度到任何节点上执行。为了防止这种情况的发生，集群管理员希望能在系统管理功能中设置禁止Pod申请超过2GB内存。
- 集群由同一个组织中的两个团队共享，各自分别用来运行生产环境和开发环境。生产环境最多可以使用8GB内存，而开发环境最多可以使用512MB内存。集群管理员希望通过为这两个环境创建不同的命名空间（namespace）并为每个命名空间设置不同的限制来满足这个需求。
- 用户创建Pod时使用的资源可能会刚好比整个机器资源的上限稍小一点，而恰好剩下的资源大小非常尴尬：不足以运行其他任务

但整个集群加起来又非常浪费。因此，集群管理员希望设置每个Pod必须至少使用集群平均资源值（CPU和内存）的20%，这样集群能够提供更好的资源一致性的调度，从而减少了资源浪费。

针对这些需求，Kubernetes提供了LimitRange机制对Pod和容器的Requests和Limits配置进一步做出限制。在下面的示例中，将说明如何将LimitsRange应用到一个Kubernetes的命名空间（namespace）中，然后说明LimitRange的几种限制方式，比如最大及最小范围、Requests和Limits的默认值、Limits与Requests最大比例上限等。

下面通过LimitRange的设置和应用对其进行说明。

1) 创建一个namespace

创建一个名为limit-example的namespace:

```
$ kubectl create namespace limit-example
namespace "limit-example" created
```

2) 为namespace设置LimitRange

为namespace“limit-example”创建一个简单的LimitRange。创建limits.yaml配置文件，内容如下:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mylimits
spec:
```

```
limits:
- max:
    cpu: "4"
    memory: 2Gi
  min:
    cpu: 200m
    memory: 6Mi
  maxLimitRequestRatio:
    cpu: 3
    memory: 2
  type: Pod
- default:
    cpu: 300m
    memory: 200Mi
  defaultRequest:
    cpu: 200m
    memory: 100Mi
  max:
    cpu: "2"
    memory: 1Gi
  min:
    cpu: 100m
    memory: 3Mi
  maxLimitRequestRatio:
    cpu: 5
    memory: 4
  type: Container
```

创建该LimitRange:

```
$ kubectl create -f limits.yaml --namespace=limit-  
example  
limitrange "mylimits" created
```

查看namespacelimit-example中的LimitRange:

```
$ kubectl describe limits mylimits --namespace=limit-  
example  
Name:      mylimits  
Namespace: limit-example  
Type  
Request    Default Limit      Min      Max      Default  
           Ratio  
-----  
-----  
Pod        3          cpu      200m     4        -  
Pod        2          memory   6Mi      2Gi      -  
Container  5          cpu      100m     2        200m300m  
Container  4          memory   3Mi      1Gi      100Mi200Mi
```

下面解释一下LimitRange中各项配置的意义和特点。

(1) 不论是CPU还是内存，在LimitRange中，Pod和Container都可以设置Min、Max和Max Limit/Requests Ratio这三种参数。Container还可以设置Default Request和Default Limit这两种参数，而Pod不能设置Default Request和Default Limit。

(2) 对Pod和Container的五种参数的解释如下。

- Container的Min（上面的100m和3Mi）是Pod中所有容器的Requests值的下限；Container的Max（上面的2和1Gi）是Pod中所有容器的Limits值的上限；Container的Default Request（上面的200m和100Mi）是Pod中所有未指定Requests值的容器的默认Requests值；Container的Default Limit（上面的300m和200Mi）是Pod中所有未指定Limits值的容器的默认Limits值。对于同一资源类型，这4个参数必须满足以下关系： $\text{Min} \leq \text{Default Request} \leq \text{Default Limit} \leq \text{Max}$ 。
- Pod的Min（上面的200m和6Mi）是Pod中所有容器的Requests值的总和的下限；Pod的Max（上面的4和2Gi）是Pod中所有容器的Limits值的总和的上限。当容器未指定Requests值或者Limits值时，将使用Container的Default Request值或者Default Limit值。
- Container的Max Limit/Requests Ratio（上面的5和4）限制了Pod中所有容器的Limits值与Requests值的比例上限；而Pod的Max Limit/Requests Ratio（上面的3和2）限制了Pod中所有容器的Limits值总和与Requests值总和的比例上限。

(3) 如果设置了Container的Max，那么对于该类资源而言，整个集群中的所有容器都必须设置Limits，否则将无法成功创建。Pod内的容器未配置Limits时，将使用Default Limit的值（本例中的300m CPU和200Mi内存），而如果Default也未配置则无法成功创建。

(4) 如果设置了Container的Min，那么对于该类资源而言，整个进群中的所有容器都必须设置Requests。如果创建Pod的容器时未配置该类资源的Requests，那么创建过程会报验证错误。Pod里容器的Requests在未配置时，可以使用默认值defaultRequest（本例中的200m CPU和100Mi内存）；如果未配置而又没有defaultRequest，那么会默认等于该容器的Limits；如果此时Limits也未定义，那么就会报错。

(5) 对于任意一个Pod而言，该Pod中所有容器的Requests总和必须大于或等于6Mi，而且所有容器的Limits总和必须小于或等于1Gi；同样，所有容器的CPU Requests总和必须大于或等于200m，而且所有容器的CPU Limits总和必须小于或等于2。

(6) Pod里任何容器的Limits与Requests的比例不能超过Container的Max Limit/Requests Ratio；Pod里所有容器的Limits总和与Requests的总和的比例不能超过Pod的Max Limit/Requests Ratio。

3) 创建Pod时触发LimitRange限制

最后，让我们看看LimitRange生效时对容器的资源限制效果。

命名空间中的限制（LimitRange）只会在Pod创建或者更新的时候执行检查。如果手动修改限制（LimitRange）为一个新的值，那么这个新的值不会去检查或限制之前已经在该命名空间中创建好的Pod。

如果用户创建Pod时，配置的资源值（CPU或者内存）超过了LimitRange的限制，那么该创建过程会报错，在错误信息中会说明详细的错误原因。

下面通过创建一个单容器Pod来展示默认限制是如何配置到Pod上的:

```
$ kubectl run nginx --image=nginx --replicas=1 --  
namespace=limit-example  
deployment "nginx" created
```

查看已创建的Pod:

```
$ kubectl get pods --namespace=limit-example
```

	NAME		READY		STATUS
RESTARTS	AGE				
	nginx-2040093540-s8vzu	1/1		Running	0

11s

查看该Pod的resources相关信息:

```
$ kubectl get pods nginx-2040093540-s8vzu --  
namespace=limit-example -o yaml | grep resources -C 8  
resourceVersion: "57"  
selfLink: /api/v1/namespaces/limit-  
example/pods/nginx-2040093540-ivimu  
uid: 67b20741-f53b-11e5-b066-64510658e388  
spec:  
  containers:  
  - image: nginx  
    imagePullPolicy: Always
```

```
name: nginx
resources:
  limits:
    cpu: 300m
    memory: 200Mi
  requests:
    cpu: 200m
    memory: 100Mi
terminationMessagePath: /dev/termination-log
volumeMounts:
```

由于该 Pod 未配置资源 Requests 和 Limits，所以使用了 namespace/limit-example 中的默认 CPU 和内存定义的 Requests 和 Limits 值。

下面创建一个超出资源限制的 Pod（使用 3CPU）：

```
invalid-pod.yaml:
apiVersion: v1
kind: Pod
metadata:
  name: invalid-pod
spec:
  containers:
  - name: kubernetes-serve-hostname
    image: gcr.io/google_containers/serve_hostname
    resources:
```



```
limits:
  cpu: "3"
  memory: 100Mi
```

创建该Pod，可以看到系统报错了，并且提供了错误原因为超过了限制。

```
$ kubectl create -f invalid-pod.yaml --
namespace=limit-example
```

```
Error from server: error when creating "invalid-
pod.yaml": Pod "invalid-pod" is forbidden: [Maximum cpu
usage per Pod is 2, but limit is 3., Maximum cpu usage per
Container is 2, but limit is 3.]
```

接下来的例子展示了LimitRange对maxLimitRequestRatio的限制：

```
limit-test-nginx.yaml:
apiVersion: v1
kind: Pod
metadata:
  name: limit-test-nginx
  labels:
    name: limit-test-nginx
spec:
  containers:
  - name: limit-test-nginx
    image: nginx
```

```
resources:
  limits:
    cpu: "1"
    memory: 512Mi
  requests:
    cpu: "0.8"
    memory: 250Mi
```

由于limit-test-nginx这个Pod的全部内存Limits总和与Requests总和的比例为 512:250，大于 LimitRange 中定义的 Pod 的内存 maxLimitRequestRatio值2，因此创建会失败：

```
$ kubectl create -f limit-test-nginx.yaml --
namespace=limit-example

Error from server: error when creating "limit-test-
nginx.yaml": pods "limit-test-nginx" is forbidden: [memory
max limit to request ratio per Pod is 2, but provided ratio
is 2.048000.]
```

下面的例子为满足LimitRange限制的Pod:

```
valid-pod.yaml:
apiVersion: v1
kind: Pod
metadata:
  name: valid-pod
  labels:
```

```
    name: valid-pod
spec:
  containers:
  - name: kubernetes-serve-hostname
    image: gcr.io/google_containers/serve_hostname
    resources:
      limits:
        cpu: "1"
        memory: 512Mi
```

创建Pod将会成功:

```
$ kubectl create -f valid-pod.yaml --namespace=limit-example
pod "valid-pod" created
```

查看该Pod的资源信息:

```
$ kubectl get pods valid-pod --namespace=limit-example
-o yaml | grep -C 6 resources
  uid: 3b1bfd7a-f53c-11e5-b066-64510658e388
spec:
  containers:
  - image: gcr.io/google_containers/serve_hostname
    imagePullPolicy: Always
    name: kubernetes-serve-hostname
    resources:
```

```
limits:
  cpu: "1"
  memory: 512Mi
requests:
  cpu: "1"
  memory: 512Mi
```

可以看到该Pod配置了明确的Limits和Requests，因此该Pod不会使用namespacelimit-example中定义的default和defaultRequest。

需要注意的是，CPU Limits强制配置这个选项在Kubernetes集群中默认是开启的；除非集群管理员在部署kubelet时，通过设置参数--cpu-cfs-quota=false来关闭该限制：

```
$ kubelet --help
Usage of kubelet
....
--cpu-cfs-quota[=true]: Enable CPU CFS quota
enforcement for containers that specify CPU limits
$ kubelet --cpu-cfs-quota=false ...
```

如果集群管理员希望对整个集群中容器或者Pod配置的Requests和Limits做限制，那么可以通过配置Kubernetes的命名空间（namespace）上的LimitRange（资源限制区间）来达到该目的。在Kubernetes集群中，如果Pod没有显式定义Limits和Requests，那么Kubernetes系统会将该Pod所在的命名空间中定义的LimitRange的default和defaultRequests配置到该Pod上。

3. 资源的服务质量管理（ResourceQoS）

本节对Kubernetes如何根据Pod的Requests和Limits配置来实现针对Pod的不同级别的资源服务质量控制（QoS）进行说明。

在Kubernetes的资源QoS体系中，需要保证高可靠性的Pod可以申请可靠资源，而一些不需要高可靠性的Pod可以申请可靠性较低或者不可靠的资源。在计算资源一节中，我们讲到了容器的资源配置分为Requests和Limits，其中Requests是Kubernetes调度时能为容器提供的完全可保障的资源量（最低保障），而Limits是系统允许容器运行时可能使用到的资源量的上限（最高上限）。Pod级别的资源配置是通过计算Pod内所有容器的资源配置的总和得出来的。

Kubernetes中Pod的Requests和Limits资源配置有如下特点：如果Pod配置的Requests值等于Limits值，那么该Pod可以获得的资源是完全可靠的；而如果Pod的Requests值小于Limits值，那么该Pod获得的资源可分成两部分：一部分是完全可靠的资源，资源量大小等于Requests值；另外一部分是不可靠的资源，这部分资源最大等于Limits与Requests的差额值，这份不可靠的资源能够申请到多少，则取决于当时主机上容器可用资源的余量。

通过这种机制，Kubernetes可以实现节点资源的超售（Over Subscription），比如在CPU完全充足的情况下，某机器共有32GiB内存可提供给容器使用，容器配置为Requests值1GiB，Limits值为2GiB，那么该机器上最多可以同时运行32个容器，每个容器最多可使用2GiB内存，如果这些容器的内存使用峰值错开，那么所有容器也可以一直正常运行。

超售机制能有效地提高资源的利用率，同时不会影响容器申请的完全可靠资源的可靠性。

1) Requests和Limits对不同计算资源类型的限制机制

根据计算资源章节的内容我们知道，容器的资源配置满足以下两个条件。

- $Requests \leq \text{节点可用资源}$ 。
- $Requests \leq Limits$ 。

Kubernetes根据Pod配置的Requests值来调度Pod，Pod在成功调度之后会得到Requests值定义的资源来运行；而如果Pod所在机器上的资源有空余，则Pod可以申请更多的资源，最多不能超过Limits的值。我们下面看一下Requests和Limits针对不同计算资源类型的限制机制的差异。这种差异主要取决于计算资源类型是可压缩资源还是不可压缩资源。

(1) 可压缩资源

- Kubernetes目前支持的可压缩资源是CPU。
- Pod可以得到Pod的Requests配置的CPU使用量，而是否能使用超过Requests值的部分取决于系统的负载和调度。不过由于目前Kubernetes和Docker的CPU隔离机制都是在容器级别隔离的，所以Pod级别的资源配置并不能完全得到保障；Pod级别的cgroups正在紧锣密鼓地开发中，如果将来引入，就可以确保Pod级别的资源配置准确运行。
- 空闲CPU资源按照容器Requests值的比例分配。举例说明：容器A的CPU配置为Requests 1Limits 10，容器B的CPU配置为request

2Limits 8，A和B同时运行在一个节点上，初始状态下容器的可用CPU为3cores，那么A和B恰好得到它们的Requests中定义的CPU用量，即1CPU和2CPU。如果A和B都需要更多的CPU资源，而恰好此时系统的其他任务释放出1.5CPU，那么这1.5CPU将按照A和B的Requests值的比例1:2分配给A和B，即最终A可使用1.5CPU，B可使用3CPU。

- 如果Pod使用了超过Limits 10中配置的CPU用量，那么cgroups会对Pod中的容器的CPU使用进行限流（throttled）；如果Pod没有配置Limits 10，那么Pod会尝试抢占所有空闲的CPU资源（Kubernetes从1.2版本开始默认开启--cpu-cfs-quota，因此默认情况下必须配置Limits）。

（2）不可压缩资源

- Kubernetes目前支持的可压缩资源是内存。
- Pod可以得到Requests中配置的内存。如果Pod使用的内存量小于它的Requests的配置，那么这个Pod可以正常运行（除非出现操作系统级别的内存不足等严重问题）；如果Pod使用的内存量超过了它的Requests的配置，那么这个Pod有可能被Kubernetes“杀掉”：比如Pod A使用了超过Requests而不到Limits的内存量，此时同一机器上另外一个Pod B之前只使用了远少于自己的Requests值的内存，而此时程序压力增大，Pod B向系统申请的总量不超过自己的Requests值的内存，那么Kubernetes可能会直接杀掉Pod A；另外一种情况是Pod A使用了超过Requests而不到Limits的内存量，此时Kubernetes将一个新的Pod调度到这台机器上，新的Pod需要使用内存，而只有Pod A使用了超过了自己的Requests值的内存，那么Kubernetes也可能会杀掉Pod A来释放内存资源。

- 如果Pod使用的内存量超过了它的Limits设置，那么操作系统内核会杀掉Pod所有容器的所有进程中使用内存最多的一个，直到内存不超过Limits为止。

2) 对调度策略的影响

- Kubernetes的kubelet通过计算Pod中所有容器的Requests的总和来决定对Pod的调度。
- 不管是CPU还是内存，Kubernetes调度器和kubelet都会确保节点上所有Pod的Requests的总和不会超过该节点上可分配给容器使用的资源容量上限。

3) 服务质量等级 (QoS Classes)

在一个超用 (Over Committed, 即容器Limits总和大于系统容量上限) 系统中，由于容器负载的波动可能导致操作系统的资源不足，最终可能会导致部分容器被“杀掉”。在这种情况下，我们当然会希望优先“杀掉”那些不太重要的容器，那么如何衡量重要程度呢？Kubernetes将容器划分成3个QoS等级：Guaranteed（完全可靠的）、Burstable（弹性波动、较可靠的）和Best-Effort（尽力而为、不太可靠的），这三种优先级依次递减，如图5.4所示。

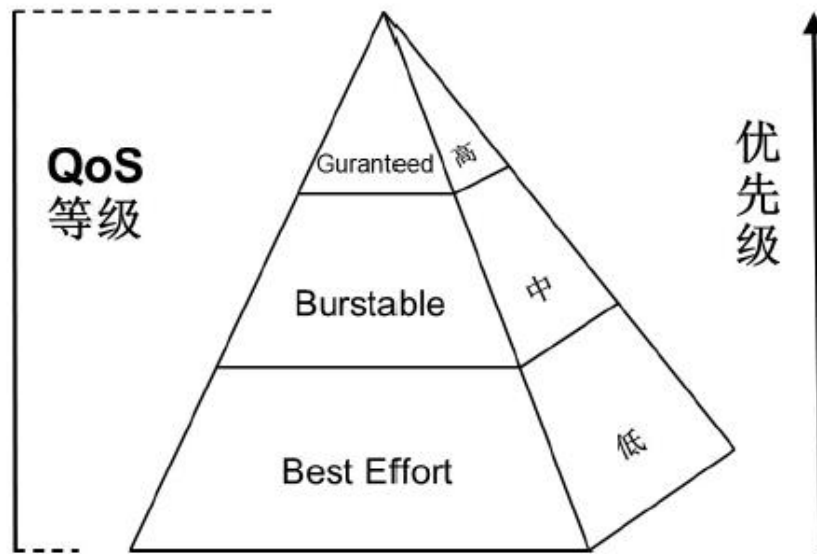


图5.4 QoS等级和优先级的关系

从理论上来说，QoS级别应该作为一个单独的参数来提供API，并由用户对Pod进行配置，这种配置应该与Requests和Limits无关。但在当前版本的Kubernetes的设计中，为了简化模式及避免引入太多的复杂性，QoS级别直接由Requests和Limits来定义。在Kubernetes中容器的QoS级别等于容器所在Pod的QoS级别，而Kubernetes的资源配置定义了Pod的三种QoS级别，如下所述。

1) Guaranteed（完全可靠的）

如果Pod中的所有容器对所有资源类型都定义了Limits和Requests，并且所有容器的Limits值都和Requests值全部相等（且都不为0），那么该Pod的QoS级别就是Guaranteed。注意：在这种情况下，容器可以不定义Requests，因为Requests值在未定义的时候默认等于Limits。

下面这两个例子中定义的Pod QoS级别就是Guaranteed:

```
containers:
  name: foo
  resources:
    limits:
      cpu: 10m
      memory: 1Gi
  name: bar
  resources:
    limits:
      cpu: 100m
      memory: 100Mi
```

在上面的例子中未定义Requests值，所以其默认等于Limits值。而下面这个例子中定义的Requests和Limits的值完全相同:

```
containers:
  name: foo
  resources:
    limits:
      cpu: 10m
      memory: 1Gi
    requests:
      cpu: 10m
      memory: 1Gi
  name: bar
```

```
resources:
  limits:
    cpu: 100m
    memory: 100Mi
  requests:
    cpu: 10m
    memory: 1Gi
```

2) Best-Effort (尽力而为、不太可靠的)

如果Pod中所有容器都未定义资源配置（Requests和Limits都未定义），那么该Pod的QoS级别就是Best-Effort。

例如下面这个Pod定义：

```
containers:
  name: foo
  resources:
  name: bar
  resources:
```

3) Burstable (弹性波动、较可靠的)

当一个Pod既不是Guaranteed级别的，也不是Best-Effort级别的时，该Pod的QoS级别就是Burstable。Burstable级别的Pod包括两种情况。第1种情况是：Pod中的一部分容器在一种或多种资源类型的资源配置中，定义了Requests值和Limits值（都不为0），且Requests值小于Limits值；第2种情况是：Pod中的一部分容器未定义资源配置

（Requests和Limits都未定义）。注意：容器未定义Limits时，Limits值默认等于节点资源容量上限。

下面几个例子中的Pod的QoS等级都是Burstable。

（1）容器foo的CPU Requests不等于Limits:

```
containers:
  name: foo
  resources:
    limits:
      cpu: 10m
      memory: 1Gi
    requests:
      cpu: 5m
      memory: 1Gi
  name: bar
  resources:
    limits:
      cpu: 10m
      memory: 1Gi
    requests:
      cpu: 10m
      memory: 1Gi
```

（2）容器bar未定义资源配置而容器foo定义了资源配置:

```
containers:
  name: foo
  resources:
    limits:
      cpu: 10m
      memory: 1Gi
    requests:
      cpu: 10m
      memory: 1Gi
  name: bar
```

(3) 容器foo未定义CPU，而容器bar未定义内存:

```
containers:
  name: foo
  resources:
    limits:
      memory: 1Gi
  name: bar
  resources:
    limits:
      cpu: 100m
```

(4) 容器bar未定义资源配置，而容器foo未定义Limits值:

```
containers:
  name: foo
```

```
resources:
  requests:
    cpu: 10m
    memory: 1Gi
name: bar
```

4) Kubernetes QoS的工作特点

Pod的CPU Requests无法得到满足（比如节点的系统级任务占用过多的CPU导致无法分配给足够的CPU给容器使用）时，容器得到的CPU会被压缩限流。

内存由于是不可压缩资源，所以针对内存资源紧缺的情况，将按照以下逻辑进行处理。

(1) **Best-Effort Pod**的优先级最低，这类Pod中运行的进程会在系统内存紧缺时被第一优先“杀死”。当然，从另外一个角度来看，**Best-Effort Pod**由于没有设置资源Limits，所以在资源充足的时候，它们可以充分地使用所有的闲置资源。

(2) **Burstable Pod**的优先级居中，这类Pod初始时会分配较少的可靠资源，但可以按需申请更多的资源。当然，如果整个系统内存紧缺，而又没有**Best-Effort**容器可以被“杀死”以释放资源，则这类Pod中的进程可能会被“杀死”。

(3) **Guaranteed Pod**的优先级最高，而且一般情况下这类Pod只要不超过其资源Limits的限制就不会被“杀死”。当然，如果整个系统内存紧缺，而又没有其他更低优先级的容器可以被“杀死”以释放资源，这类Pod中的进程也可能被“杀死”。

5) OOM计分系统

OOM (Out Of Memory) 计分规则包括如下内容。

- OOM计分是一个进程消耗内存在系统中占的百分比中不含百分号的数字的值乘以10的结果，这个结果是进程OOM基础分；将进程OOM基础分的分值再加上这个进程的OOM分数调整值OOM_SCORE_ADJ的值作为进程OOM最终分值（除root启动的进程外）。在系统发生OOM时，OOM Killer会优先杀掉OOM计分更高的进程。
- 进程的OOM计分的基本分数值范围是0~1000，如果A进程的调整值OOM_SCORE_ADJ减去B进程的调整值的结果大于1000，那么A进程的OOM计分最终值必然大于B进程，A进程会比B进程优先被杀死。
- 不论调整值OOM_SCORE_ADJ为多少，任何进程的最终分值范围也是0~1000。

在Kubernetes，不同QoS的OOM计分调整值规则如表5.1所示。

表5.1 不同QoS的OOM计分调整值

QoS 等级	oom_score_adj
Guaranteed	-998
BestEffort	1000
Burstable	$\min(\max(2, 1000 - (1000 * \text{memoryRequestBytes}) / \text{machineMemoryCapacityBytes}), 999)$

- Best-effort Pod设置OOM_SCORE_ADJ调整值为1000，因此Best-effort Pod中的容器里面的所有进程的OOM最终分肯定是1000。

- **Guaranteed Pod** 设置 `OOM_SCORE_ADJ` 调整值为 `-998`，因此 **Guaranteed Pod** 中的容器里面的所有进程的 **OOM** 最终分一般为 `0` 或者 `1`（因为基础分不可能为 `1000`）。
- **Burstable Pod** 规则分情况说明：如果 **Burstable Pod** 的内存 `Requests` 超过了系统可用内存的 `99.8%`，那么这个 **Pod** 的 `OOM_SCORE_ADJ` 调整值固定为 `2`；否则，设置 `OOM_SCORE_ADJ` 调整值为 `1000 - 10 * (内存Requests占系统可用内存的百分比的无百分号的数字部分)` 的值），而如果内存 `Requests` 为 `0`，那么 `OOM_SCORE_ADJ` 调整值固定为 `999`。这样的规则能确保 `OOM_SCORE_ADJ` 调整值的范围为 `2~999`，而 **Burstable Pod** 中所有进程的 **OOM** 最终分数范围为 `2~1000`。**Burstable Pod** 进程的 **OOM** 最终分数始终大于 **Guaranteed Pod** 的进程得分，因此它们会被优先“杀死”。如果一个 **Burstable Pod** 使用的内存比它的内存 `Requests` 少，那么可以肯定的是它的所有进程的 **OOM** 最终分数会小于 `1000`，此时能确保它的优先级高于 **Best-effort Pod**。如果一个 **Burstable Pod** 的某个容器中某个进程使用的内存比容器的 `request` 值高，那么这个进程的 **OOM** 最终分数会是 `1000`，否则它的 **OOM** 最终分会小于 `1000`。假设下面容器中有一个占用内存非常大的进程，那么当一个使用内存超过其 `Requests` 的 **Burstable Pod** 与另外一个使用内存少于其 `Requests` 的 **Burstable Pod** 发生内存竞争冲突时，前者的进程会被系统“杀掉”。如果一个 **Burstable Pod** 内部有多个进程的多个容器发生内存竞争冲突，那么此时 **OOM** 评分只能作为参考，不能保证完全按照资源配置的定义来执行 **OOM Kill**。

OOM 还有一些特殊的计分规则，如下所述。

- **kubelet** 进程和 **Docker** 进程的调整值 `OOM_SCORE_ADJ` 为 `-998`。

- 如果配置进程调整值OOM_SCORE_ADJ为-999，那么这类进程不会被OOM Killer“杀掉”。

6) QoS的演进

目前Kubernetes基于QoS的超用机制日趋完善，但还有一些问题需要解决。

7) 内存Swap的支持

当前的QoS策略都是假定主机不启用内存Swap。如果主机启用了Swap，那么上面的QoS策略可能会失效。举例说明：两个Guaranteed Pod都刚好达到了内存Limits，那么由于内存Swap机制，它们还可以继续申请使用更多的内存。如果Swap空间不足，那么最终这两个Pod中的进程可能会被“杀掉”。由于Kubernetes和Docker尚不支持内存Swap空间的隔离机制，所以这一功能暂时还未实现。

8) 更丰富的QoS策略

当前的QoS策略都是基于Pod的资源配置（Requests和Limits）来定义的，而资源配置本身又承担着对Pod资源管理和限制的功能。两种不同维度的功能使用同一个参数来配置，可能会导致某些复杂需求无法满足，比如当前Kubernetes无法支持弹性的、高优先级的Pod。自定义QoS优先级能提供更大的灵活性，完美地实现各类需求，但同时会引入更高的复杂性，而且过于灵活的设置会给予用户过高的权限，对系统管理也提出了更大的挑战。

4.资源的配额管理（Resource Quotas）

如果一个Kubernetes集群被多个用户或者多个团队共享使用，那么就需要考虑共享时对资源公平使用的问题，因为某个用户可能会使用超过基于公平原则分配给其的资源量。

资源配额（Resource Quotas）就是解决这个问题的工具。通过ResourceQuota对象，我们可以定义一项资源配额，这个资源配额可以为每一个命名空间（namespace）提供一个总体的资源使用的限制：它可以限制命名空间中某种类型的对象的总数目上限，也可以设置命名空间中Pod可以使用到的计算资源的总上限。

典型的资源配额（Resource Quotas）使用方式如下。

- 不同的团队工作在不同的命名空间下，目前这个是非约束性的，未来版本中可能会通过ACLs（访问控制列表Access Control List）的方式来实现强制性约束。
- 集群管理员为集群中的每个命名空间创建一个或者多个资源配额项。
- 当用户在命名空间中使用资源（创建Pod或者Service等）时，Kubernetes的配额系统会统计、监控和检查资源用量，以确保使用的资源用量没有超过资源配额的配置。
- 如果创建或者更新应用时，资源使用超过了某项资源配额的限制，那么创建或者更新的请求会报错（HTTP 403Forbidden），错误信息给出详细的出错原因说明。
- 如果命名空间中的计算资源（CPU和内存）的资源配额启用，那么用户必须为相应的资源类型设置Requests或Limits；否则配额系统可能会直接拒绝Pod的创建。这里可以使用LimitRange机制来为没有配置资源的Pod提供默认资源配置。

下面的例子展示了一个非常适合使用资源配额来做资源控制管理的场景。

- 集群共有32GB内存和16CPU，两个小组，A小组使用20GB内存和10CPU，B小组使用10GB内存和2CPU，剩下的2GB内存和2CPU作为预留。
- 在名为testing的命名空间中，限制使用1CPU和1GB内存；在名为production的命名空间中，资源使用不受限制。

在使用资源配额时，需要注意以下两点。

- 如果集群中总的可用资源小于各命名空间中资源配额的总和，那么可能会导致资源竞争。资源竞争时，Kubernetes系统使用先到先得的原则。
- 不管是资源竞争还是配额的修改都不会影响到已经创建的资源使用对象。

1) 在Master中开启资源配额选型

资源配额可以通过在kube-apiserver的--admission-control=参数值中添加ResourceQuota参数进行开启。如果某个命名空间的定义中存在ResourceQuota，那么对于该命名空间而言，资源配额就是开启的。一个命名空间可以有多个ResourceQuota配置项。

(1) 计算资源配额 (Compute Resource Quota)

资源配额可以限制一个命名空间中所有Pod的计算资源的总和。表5.2列出了目前Kubernetes资源配额支持限制的計算资源类型。

表5.2 ResourceQuota的计算资源类型

资源名称	说明
cpu	所有非终止状态的 Pod，CPU Requests 的总和不能超过该值
limits.cpu	所有非终止状态的 Pod，CPU Limits 的总和不能超过该值
limits.memory	所有非终止状态的 Pod，内存 Limits 的总和不能超过该值
memory	所有非终止状态的 Pod，内存 Requests 的总和不能超过该值
requests.cpu	所有非终止状态的 Pod，CPU Requests 的总和不能超过该值
requests.memory	所有非终止状态的 Pod，内存 Requests 的总和不能超过该值

(2) 对象数量配额 (Object Count Quota)

指定类型的对象数量可以被限制。表5.3列出了Kubernetes资源配额支持限制对象数量的对象类型。

表5.3 ResourceQuota的对象类型

资源名称	说明
configmaps	在该命名空间中，能存在的 ConfigMap 的总数上限
persistentvolumeclaims	在该命名空间中，能存在的持久卷的总数上限
pods	在该命名空间中，能存在的非终止状态 Pod 的总数上限。Pod 终止状态等价于 Pod 的 status.phase 状态值为 Failed 或者 Succeeded
replicationcontrollers	在该命名空间中，能存在的 RC 的总数上限
resourcequotas	在该命名空间中，能存在的资源配额项 (ResourceQuota) 的总数上限
services	在该命名空间中，能存在的 service 的总数上限
services.loadbalancers	在该命名空间中，能存在的负载均衡 (LoadBalancer) 的总数上限
services.nodeports	在该命名空间中，能存在的 NodePort 的总数上限
secrets	在该命名空间中，能存在的 Secret 的总数上限

例如我们可以通过资源配额来限制命名空间中能创建的Pod的最大数量。这种设置可以防止某些用户大量创建Pod而迅速耗尽整个集群的Pod IP和计算资源。

2) 配额的作用域 (Quota Scopes)

每项资源配额都可以单独配置一组作用域，配置了作用域的资源配额只会对符合其作用域的资源使用进行计量和限制，作用域范围内

的且超过了资源配额的请求都会报验证错。表5.4列出了ResourceQuota的4种作用域。

表5.4 ResourceQuota的作用域

作用域	说明
Terminating	匹配所有 spec.activeDeadlineSeconds >= 0 的 Pod
NotTerminating	匹配所有 spec.activeDeadlineSeconds 是 nil 的 Pod
BestEffort	匹配所有 QoS 为 Best-Effort 的 Pod
NotBestEffort	匹配所有 QoS 不是 Best-Effort 的 Pod

其中，BestEffort作用域可以限定资源配额来追踪pods资源的使用，Terminating、NotTerminating和NotBestEffort这三种作用域可以限定资源配额来追踪以下资源的使用。

- cpu
- limits.cpu
- limits.memory
- memory
- pods
- requests.cpu
- requests.memory

3) 在资源配额（ResourceQuota）中设置Requests和Limits

资源配额也可以设置Requests和Limits。

如果资源配额中指定了requests.cpu或requests.memory，那么它会强制要求每一个容器都必须配置自己的CPU Requests或CPU Limits（可使用LimitRange提供的默认值）。

同理，如果资源配额中指定了limits.cpu或limits.memory，那么它也会强制要求每一个容器都必须配置自己的内存Requests或内存Limits（可使用LimitRange提供的默认值）。

4) 资源配额（ResourceQuota）的定义

下面通过几个例子对资源配额进行设置和应用。

与LimitRange相似，ResourceQuota也设置在namespace中。创建名为myspace的namespace:

```
$ kubectl create namespace myspace
namespace "myspace" created
```

创建ResourceQuota配置文件compute-resources.yaml，用于设置计算资源的配额:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    pods: "4"
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

创建该项资源配额:

```
$ kubectl create -f compute-resources.yaml --  
namespace=myspace  
resourcequota "compute-resources" created
```

创建另一个名为object-counts.yaml的文件，用于设置对象数量的配额:

```
apiVersion: v1  
kind: ResourceQuota  
metadata:  
  name: object-counts  
spec:  
  hard:  
    configmaps: "10"  
    persistentvolumeclaims: "4"  
    replicationcontrollers: "20"  
    secrets: "10"  
    services: "10"  
    services.loadbalancers: "2"
```

创建该ResourceQuota:

```
$ kubectl create -f object-counts.yaml --  
namespace=myspace  
resourcequota "object-counts" created
```

查看各ResourceQuota的详细信息:

```
$ kubectl describe quota compute-resources --  
namespace=myspace
```

Name:	compute-resources	
Namespace:	myspace	
Resource	Used	Hard
-----	----	----
limits.cpu	0	2
limits.memory	0	2Gi
Pods	0	4
requests.cpu	0	1
requests.memory	0	1Gi

```
$ kubectl describe quota object-counts --  
namespace=myspace
```

Name:	object-counts	
Namespace:	myspace	
Resource	Used	Hard
-----	----	----
configmaps	0	10
persistentvolumeclaims	0	4
replicationcontrollers	0	20
secrets	1	10
services	0	10
services.loadbalancers	0	2

5) 资源配额与集群资源总量的关系

资源配额与集群资源总量是完全独立的。资源配额是通过绝对的单位来配置的：这也就意味着如果集群中新添加了节点，那么资源配额不会自动更新，而该资源配额所对应的命名空间下对象也不能自动地增加资源上限。

在某些情况下，我们可能希望资源配额能支持更复杂的策略，如下所述。

- 对于不同的租户，按照比例划分整个集群的资源。
- 允许每个租户都能按照需要来提高资源用量，但是有一个较宽容的限制，以防止意外的资源耗尽情况发生。
- 探测某个命名空间的需求，添加物理节点并扩大资源配额值。

这些策略可以通过将资源配额作为一个控制模块、手动编写一个控制器（**controller**）来监控资源使用情况，并调整命名空间上的资源配额的方式来实现。

资源配额将整个集群中的资源总量做了一个静态的划分，但它并没有对集群中的节点（**Node**）做任何限制：不同命名空间中的**Pod**仍然可以运行到同一个节点上。

5.ResourceQuota和LimitRange实践指南

根据前面对资源管理的介绍，这里将通过一个完整的例子来说明如何通过资源配额和资源配置范围的配合来控制一个命名空间的资源使用。

集群管理员根据集群用户数量来调整集群配置，以达到如下目的：能控制特定命名空间中的资源使用量，最终实现集群的公平使用和成本的控制。

需要实现的功能如下。

- 限制运行状态的Pod的计算资源用量。
- 限制持久存储卷的数量以控制对存储的访问。
- 限制负载均衡器的数量以控制成本。
- 防止滥用网络端口这类稀缺资源。
- 提供默认的计算资源Requests以便于系统做出更优化的调度。

1) 创建命名空间

创建名为quota-example的命名空间，namespace.yaml文件的内容如下：

```
apiVersion: v1
kind: Namespace
metadata:
  name: quota-example

$ kubectl create -f namespace.yaml
namespace "quota-example" created
```

查看命名空间：

```
$ kubectl get namespaces
```

NAME	STATUS	AGE
default	Active	2m
kube-system	Active	2m
quota-example	Active	39s

2) 设置限定对象数目的资源配额

通过设置限定对象的数量资源配额，可以控制以下资源的数量：

- 持久存储卷；
- 负载均衡器；
- NodePort。

创建名为object-counts的ResourceQuota：

```
object-counts.yaml:
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
spec:
  hard:
    persistentvolumeclaims: "2"
    services.loadbalancers: "2"
    services.nodeports: "0"
```

```
$ kubectl create -f object-counts.yaml --  
namespace=quota-example  
resourcequota "object-counts" created
```

配额系统会检测到资源项配额的创建，并且将会统计和限制该命名空间中的资源消耗。

查看该配额是否生效：

```
$ kubectl describe quota object-counts --  
namespace=quota-example
```

Name:	object-counts		
Namespace:	quota-example		
Resource	Used	Hard	
-----	----	----	
persistentvolumeclaims	0	2	
services.loadbalancers	0	2	
services.nodeports	0	0	

至此，配额系统会自动阻止那些使资源用量超过资源配额限定值的请求。

3) 设置限定计算资源的资源配额

下面我们再来创建一项限定计算资源的资源配额，以限制该命名空间中的计算资源的使用总量。

创建名为compute-resources的ResourceQuota:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    pods: "4"
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

```
$ kubectl create -f compute-resources.yaml --
namespace=quota-example
resourcequota "compute-resources" created
```

查看该配额是否生效:

```
$ kubectl describe quota compute-resources --
namespace=quota-example
```

Name:	compute-resources	
Namespace:	quota-example	
Resource	Used	Hard
-----	----	----
limits.cpu	0	2
limits.memory	0	2Gi
pods	0	4

requests.cpu	0	1
requests.memory	0	1Gi

配额系统会自动防止该命名空间下同时拥有超过4个非“终止态”的Pod。此外，由于该项资源配额限制了CPU和内存的Limits和Requests的总量，因此会强制要求该命名空间下的所有容器都必须显式地定义CPU和内存的Limits和Requests（可使用默认值，Requests默认等于Limits）。

4) 配置默认Requests和Limits

在命名空间已经配置了限定计算资源的资源配额的情况下，如果尝试在该命名空间下创建一个不指定Requests和Limits的Pod，那么Pod的创建可能会失败。下面是一个失败的例子。

创建一个Nginx的Deployment:

```
$ kubectl run nginx --image=nginx --replicas=1 --  
namespace=quota-example  
deployment "nginx" created
```

查看创建的Pod，会发现Pod没有创建成功:

```
$ kubectl get pods --namespace=quota-example
```

再查看一下Deployment的详细信息:

```
$ kubectl describe deployment nginx --namespace=quota-
example
Name:                               nginx
Namespace:                           quota-example
CreationTimestamp:                   Mon, 06 Jun 2016 16:11:37
-0400
Labels:                             run=nginx
Selector:                           run=nginx
Replicas:                           0 updated | 1 total | 0
available | 1 unavailable
StrategyType:                       RollingUpdate
MinReadySeconds:                    0
RollingUpdateStrategy:              1 max unavailable, 1 max surge
OldReplicaSets:                     <none>
NewReplicaSet:                      nginx-3137573019 (0/1 replicas
created)
...
```

本 Deployment 尝试创建一个 Pod，但是失败了，查看其中 ReplicaSet 的详细信息：

```
$ kubectl describe rs nginx-3137573019 --
namespace=quota-example
Name:                               nginx-3137573019
Namespace:                           quota-example
Image(s):                           nginx
Selector:                           pod-template-
```

```

hash=3137573019,run=nginx
    Labels:                pod-template-hash=3137573019
                           run=nginx
    Replicas:              0 current / 1 desired
    Pods Status:          0 Running / 0 Waiting / 0
Succeeded / 0 Failed
    No volumes.
    Events:
      FirstSeen    LastSeen    Count  From
SubobjectPath Type      Reason      Message
-----
-----
4m          7s          11      {replicaset-controller }
Warning      FailedCreate  Error creating: pods "nginx-
3137573019-" is forbidden: Failed quota: compute-resources:
must specify
limits.cpu,limits.memory,requests.cpu,requests.memory

```

可以看到Pod创建失败的原因：Master拒绝了这个ReplicaSet创建Pod，因为这个Pod中没有指定CPU和内存的Requests和Limits。

为了避免这种失败，我们可以使用LimitRange来为这个命名空间下的所有Pod提供一个资源配置的默认值。下面的例子展示了如何为这个命名空间添加一个指定默认资源配置的LimitRange。

创建一个名为limits的LimitRange:

```
limits.yaml:
apiVersion: v1
kind: LimitRange
metadata:
  name: limits
spec:
  limits:
  - default:
      cpu: 200m
      memory: 512Mi
    defaultRequest:
      cpu: 100m
      memory: 256Mi
    type: Container
```

```
$ kubectl create -f limits.yaml --namespace=quota-
example
```

```
limitrange "limits" created
```

```
$ kubectl describe limits limits --namespace=quota-
example
```

```
Name:          limits
Namespace:     quota-example
Type          Resource  Min   Max   Default Request
Default Limit  Max Limit/Request Ratio
-----
-----
```

Container memory	-	-	256Mi512Mi	-
Container cpu	-	-	100m200m	-

LimitRange创建成功后，用户在该命名空间下的创建未指定资源配置的**Pod**的请求时，系统会自动为该**Pod**设置默认的资源配置。

例如，每个新建的未指定资源配置的**Pod**都等价于使用下面的资源配置：

```
$ kubectl run nginx \
  --image=nginx \
  --replicas=1 \
  --requests=cpu=100m,memory=256Mi \
  --limits=cpu=200m,memory=512Mi \
  --namespace=quota-example
```

至此，我们已经为该命名空间配置好了默认的计算资源，我们的**ReplicaSet**应该能够创建**Pod**了。查看一下，创建**Pod**成功了：

```
$ kubectl get pods --namespace=quota-example
```

	NAME	READY	STATUS	RESTARTS
AGE				
	nginx-3137573019-fvrig	1/1	Running	0
6m				

接下来，还可以随时查看资源配额的使用情况：

```
$ kubectl describe quota --namespace=quota-example
```

```
Name:          compute-resources
```

```
Namespace:     quota-example
```

```
Resource       Used    Hard
```

```
-----
```

```
limits.cpu      200m    2
```

```
limits.memory   512Mi   2Gi
```

```
Pods           1        4
```

```
requests.cpu    100m    1
```

```
requests.memory 256Mi   1Gi
```

```
Name:          object-counts
```

```
Namespace:     quota-example
```

```
Resource       Used    Hard
```

```
-----
```

```
persistentvolumeclaims 0        2
```

```
services.loadbalancers  0        2
```

```
services.nodeports      0        0
```

可以看到每个Pod创建时都会消耗掉指定的资源量，而这些使用量都会被Kubernetes准确地跟踪、监控和管理。

5) 指定资源配额的作用域

假设我们并不想为某个命名空间配置默认的计算资源配额，而是希望限定在命名空间内运行的QoS为BestEffort的Pod总数，例如将集群

中的部分资源用来运行QoS为非BestEffort的服务，而将闲置的资源用来运行QoS为BestEffort的服务，即可避免集群的所有资源仅被大量的BestEffort Pod耗尽。这可以通过创建两个资源配额（ResourceQuota）来实现。

首先创建一个名为quota-scopes的命名空间：

```
$ kubectl create namespace quota-scopes
namespace "quota-scopes" created
```

创建一个名为 best-effort 的 ResourceQuota，指定 Scope 为 BestEffort：

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: best-effort
spec:
  hard:
    pods: "10"
  scopes:
    - BestEffort
```

```
$ kubectl create -f best-effort.yaml --
namespace=quota-scopes
resourcequota "best-effort" created
```

再创建一个名为 not-best-effort 的 ResourceQuota，指定 Scope 为 NotBestEffort:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: not-best-effort
spec:
  hard:
    pods: "4"
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
  scopes:
    - NotBestEffort
```

```
$ kubectl create -f not-best-effort.yaml --
namespace=quota-scopes
resourcequota "not-best-effort" created
```

查看创建成功的ResourceQuota:

```
$ kubectl describe quota --namespace=quota-scopes
Name:          best-effort
Namespace:     quota-scopes
Scopes:        BestEffort
```

* Matches all pods that have best effort quality of service.

Resource	Used	Hard
-----	-----	-----
Pods	0	10

Name: not-best-effort

Namespace: quota-scopes

Scopes: NotBestEffort

* Matches all pods that do not have best effort quality of service.

Resource	Used	Hard
-----	-----	-----
limits.cpu	0	2
limits.memory	0	2Gi
Pods	0	4
requests.cpu	0	1
requests.memory	0	1Gi

之后，对于没有配置 Requests 的 Pod 将会被名为 best-effort 的 ResourceQuota 所限制；而配置了 Requests 的 Pod 会被名为 not-best-effort 的 ResourceQuota 所限制。

创建两个 Deployment:

```
$ kubectl run best-effort-nginx --image=nginx --replicas=8 --namespace=quota-scopes
```

```
deployment "best-effort-nginx" created
```

```
$ kubectl run not-best-effort-nginx \
  --image=nginx \
  --replicas=2 \
  --requests=cpu=100m,memory=256Mi \
  --limits=cpu=200m,memory=512Mi \
  --namespace=quota-scopes
deployment "not-best-effort-nginx" created
```

名为 `best-effort-nginx` 的 Deployment 因为没有配置 `Requests` 和 `Limits`，所以它的 QoS 级别为 `BestEffort`，因此它的创建过程由 `best-effort` 资源配额项来限制，而 `not-best-effort` 资源配额项不会对它进行限制。 `best-effort` 资源配额项没有限制 `Requests` 和 `Limits`，因此 `best-effort-nginx` Deployment 可以成功地创建 8 个 Pod。

名为 `not-best-effort-nginx` 的 Deployment 因为配置了 `Requests` 和 `Limits`，且二者不相等，所以它的 QoS 级别为 `Burstable`，因此它的创建过程由 `not-best-effort` 资源配额项来限制，而 `best-effort` 资源配额项不会对它进行限制。 `not-best-effort` 资源配额项限制了 Pod 的 `Requests` 和 `Limits` 的总上限， `not-best-effort-nginx` Deployment 并没有超过这个上限，所以可以成功地创建两个 Pod。

查看已经创建的 Pod:

```
$ kubectl get pods --namespace=quota-scopes
```

	NAME	
STATUS	RESTARTS	AGE
		READY

		best-effort-nginx-3488455095-2qb41	1/1
Running	0	51s	
		best-effort-nginx-3488455095-3go7n	1/1
Running	0	51s	
		best-effort-nginx-3488455095-9o2xg	1/1
Running	0	51s	
		best-effort-nginx-3488455095-eyg40	1/1
Running	0	51s	
		best-effort-nginx-3488455095-gcs3v	1/1
Running	0	51s	
		best-effort-nginx-3488455095-rq8p1	1/1
Running	0	51s	
		best-effort-nginx-3488455095-udhhd	1/1
Running	0	51s	
		best-effort-nginx-3488455095-zmk12	1/1
Running	0	51s	
		not-best-effort-nginx-2204666826-7sl61	1/1
Running	0	23s	
		not-best-effort-nginx-2204666826-ke746	1/1
Running	0	23s	

可以看到10个Pod都创建成功。

再看一下两个资源配额项的使用情况：

```
$ kubectl describe quota --namespace=quota-scopes
```

```
Name:                best-effort
```



```
Namespace:      quota-scopes
Scopes:         BestEffort
    * Matches all pods that have best effort quality of
service.
```

Resource	Used	Hard
-----	----	----
Pods	8	10

```
Name:           not-best-effort
Namespace:      quota-scopes
Scopes:         NotBestEffort
    * Matches all pods that do not have best effort
quality of service.
```

Resource	Used	Hard
-----	----	----
limits.cpu	400m	2
limits.memory	1Gi	2Gi
Pods	2	4
requests.cpu	200m	1
requests.memory	512Mi	1Gi

可以看到best-effort资源配额项已经统计到了best-effort-nginx Deployment中创建的8个Pod的资源使用信息，而not-best-effort资源配额项也统计到了not-best-effort-nginx Deployment中创建的两个Pod的资源使用信息。

通过这个例子我们可以看到：资源配额的作用域（**Scopes**）提供了一种将资源集合分割的机制，这种机制使得集群管理员可以更加方便地监控和限制不同类型对象对于各类资源的使用，同时能为资源分配和限制提供更大的灵活度和便利性。

6. 资源管理总结

Kubernetes 中的资源管理的基础是容器和 Pod 的资源配置（**Requests**和**Limits**）。容器的资源配置（**Requests**和**Limits**）指定了容器请求的资源 and 容器能使用的资源上限，而 Pod 的资源配置则是 Pod 中所有容器的资源配置总和的上限。

通过资源配额（**Resource Quota**）机制，我们可以对命名空间下所有 Pod 使用资源的总量进行限制，也可以对这个命名空间中指定类型的对象的数量进行限制。使用作用域可以让资源配额只对符合特定范围的对象加以限制，因此作用域（**Scopes**）机制可以使资源配额的策略更加丰富灵活。

如果我们需要对用户的 Pod 或容器的资源配置做更多的限制，则我们可以使用资源配置范围（**LimitRange**）来达到这个目的。**LimitRange**可以有效地限制 Pod 和容器的资源配置的最大、最小范围，也可以限制 Pod 和容器的 **Limits** 与 **Requests** 的最大比例上限，此外 **LimitRange** 还可以为 Pod 中的容器提供默认的资源配置。

Kubernetes 基于 Pod 的资源配置（**Requests**和**Limits**）实现了资源服务质量（**QoS**）。不同 **QoS** 级别的 Pod 在系统中拥有不同的优先级：高优先级的 Pod 具有更高的可靠性，可以用于运行可靠性要求较高的服

务；而低优先级的Pod可以实现集群资源的超售，能有效地提高集群资源利用率。

上面的多种机制共同组成了当前版本Kubernetes的资源管理体系。这个资源管理体系已经可以满足大部分资源管理的需求了。同时，Kubernetes资源管理体系仍然在不停地发展和进化中，对于一些目前无法满足的更复杂、更个性化的需求，我们可以继续关注Kubernetes未来的发展和变化。

5.1.5 Kubernetes集群高可用部署方案

Kubernetes作为容器应用的管理平台，通过对Pod的运行状况进行监控，并且根据主机或容器失效的状态将新的Pod调度到其他Node上，实现了应用层的高可用性。针对Kubernetes集群，高可用性还应包含以下两个层面的考虑：etcd数据存储的高可用性和Kubernetes Master组件的高可用性。

1.etcd高可用部署

etcd在整个Kubernetes集群中处于中心数据库的地位，为保证Kubernetes集群的高可用性，首先需要保证数据库不是单故障点。一方面，etcd需要以集群的方式进行部署，以实现etcd数据存储的冗余、备份与高可用性；另一方面，etcd存储的数据本身也应考虑使用可靠的存储设备。

etcd集群的部署可以使用静态配置，也可以通过etcd提供的REST API在运行时动态添加、修改或删除集群中的成员。本节将对etcd集群的静态配置进行说明。关于动态修改的操作方法请参考etcd官方文档的说明。

首先，规划一个至少3台服务器（节点）的etcd集群，在每台服务器上安装好etcd。

部署一个由3台服务器组成的etcd集群，其配置如表5.5所示，其集群部署实例如图5.5所示。

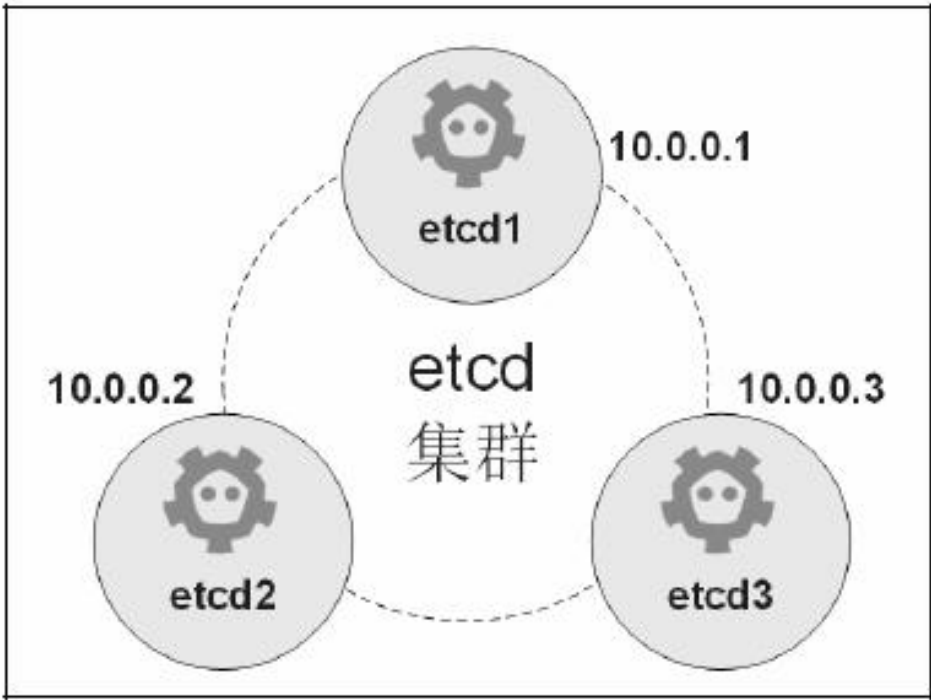


图5.5 etcd集群部署实例

表5.5 etcd集群的配置

etcd 实例名称	IP 地址
etcd1	10.0.0.1
etcd2	10.0.0.2
etcd3	10.0.0.3

然后修改每台服务器上etcd的配置文件/etc/etcd/etcd.conf。

以 etcd1 为创建集群的实例，需要将其 ETCD_INITIAL_CLUSTER_STATE 设置为“new”。etcd1 的完整配置如下：

```
# [member]

ETCD_NAME=etcd1           #etcd实例名称

ETCD_DATA_DIR="/var/lib/etcd"  #etcd数据保存目录


ETCD_LISTEN_CLIENT_URLS="http://10.0.0.1:2379,http://127.0.0.1:2379"  #供外部客户端使用的URL


ETCD_ADVERTISE_CLIENT_URLS="http://10.0.0.1:2379,http://127.0.0.1:2379"  #广播给外部客户端使用的URL

#[cluster]

ETCD_LISTEN_PEER_URLS="http://10.0.0.1:2380"  #集群内部通信使用的URL


ETCD_INITIAL_ADVERTISE_PEER_URLS="http://10.0.0.1:2380"  #广播给集群内其他成员访问的URL


ETCD_INITIAL_CLUSTER="etcd1=http://10.0.0.1:2380,etcd2=http://10.0.0.2:2380,etcd3=http://10.0.0.3:2380"  #初始集群成员列表

ETCD_INITIAL_CLUSTER_STATE="new"  #初始集群状态，new为新建集群

ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"  #集群名称
```

启动etcd1服务器上的etcd服务:

```
$ systemctl restart etcd
```

启动完成后, 就创建了一个名为etcd-cluster的集群。

etcd2 和 etcd3 为加入 etcd-cluster 集群的实例, 需要将其 ETCD_INITIAL_CLUSTER_STATE 设置为“exist”。etcd2 的完整配置如下 (etcd3 的配置略):

```
# [member]
ETCD_NAME=etcd2                #etcd实例名称
ETCD_DATA_DIR="/var/lib/etcd"  #etcd数据保存目录

ETCD_LISTEN_CLIENT_URLS="http://10.0.0.2:2379,http://127.0.0.1:2379"  #供外部客户端使用的URL

ETCD_ADVERTISE_CLIENT_URLS="http://10.0.0.2:2379,http://127.0.0.1:2379"  #广播给外部客户端使用的URL
#[cluster]
ETCD_LISTEN_PEER_URLS="http://10.0.0.2:2380"  #集群内部通信使用的URL

ETCD_INITIAL_ADVERTISE_PEER_URLS="http://10.0.0.2:2380"  #广播给集群内其他成员使用的URL

ETCD_INITIAL_CLUSTER="etcd1=http://10.0.0.1:2380,etcd2=http
```

```
://10.0.0.2:2380,etcd3=http://10.0.0.3:2380"      #初始集群成员列表
```

```
ETCD_INITIAL_CLUSTER_STATE="new"      #初始集群状态，new  
为新建集群
```

```
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"      #集群名称
```

启动etcd2和etcd3服务器上的etcd服务:

```
$ systemctl restart etcd
```

启动完成后，在任意etcd节点执行etcdctl cluster-health命令来查询集群的运行状态:

```
$ etcdctl cluster-health  
cluster is healthy  
member ce2a822cea30bfca is healthy  
member acda82ba1cf790fc is healthy  
member eba209cd0012cd2 is healthy
```

在任意etcd节点上执行etcdctl member list命令来查询集群的成员列表:

```
$ etcdctl member list  
  
ce2a822cea30bfca:      name=default  
peerURLs=http://10.0.0.1:2380,http://127.0.0.1:7001  
clientURLs=http://10.0.0.1:2379,http://127.0.0.1:2379  
  
acda82ba1cf790fc:      name=default
```



```
peerURLs=http://10.0.0.2:2380,http://127.0.0.1:7001
clientURLs=http://10.0.0.2:2379,http://127.0.0.1:2379
                                eba209cd40012cd2:      name=default
peerURLs=http://10.0.0.3:2380,http://127.0.0.1:7001
clientURLs=http://10.0.0.3:2379,http://127.0.0.1:2379
```

至此，一个etcd集群就创建成功了。

以kube-apiserver为例，将访问etcd集群的参数设置为：

```
--etcd-
servers=http://10.0.0.1:2379,http://10.0.0.2:2379,http://10
.0.0.3:2379
```

在etcd集群成功启动之后，如果需要对集群成员进行修改，则请参考官方文档的详细说明：

```
https://github.com/coreos/etcd/blob/master/Documentation/runtime-configuration.md#cluster-reconfiguration-operations。
```

对于etcd中需要保存的数据的可靠性，可以考虑使用RAID磁盘阵列、高性能存储设备、共享存储文件系统，或者使用云服务商提供的存储系统等来实现。

2.Master高可用部署

在Kubernetes系统中，Master服务扮演着总控中心的角色，主要的三个服务kube-apiserver、kube-controller-manster和kube-scheduler通过不断与工作节点上的kubelet和kube-proxy进行通信来维护整个集群的健康工作状态。如果Master的服务无法访问到某个Node，则会将该Node标记为不可用，不再向其调度新建的Pod。但对Master自身则需要进行额外的监控，使Master不成为集群的单故障点，所以对Master服务也需要进行高可用方式的部署。

以 Master 的 kube-apiserver、kube-controller-manster 和 kube-scheduler 三个服务作为一个部署单元，类似于etcd集群的典型部署配置。使用至少三台服务器安装Master服务，并且需要保证任何时候总有一套Master能够正常工作。图5.6展示了一种典型的部署方式。

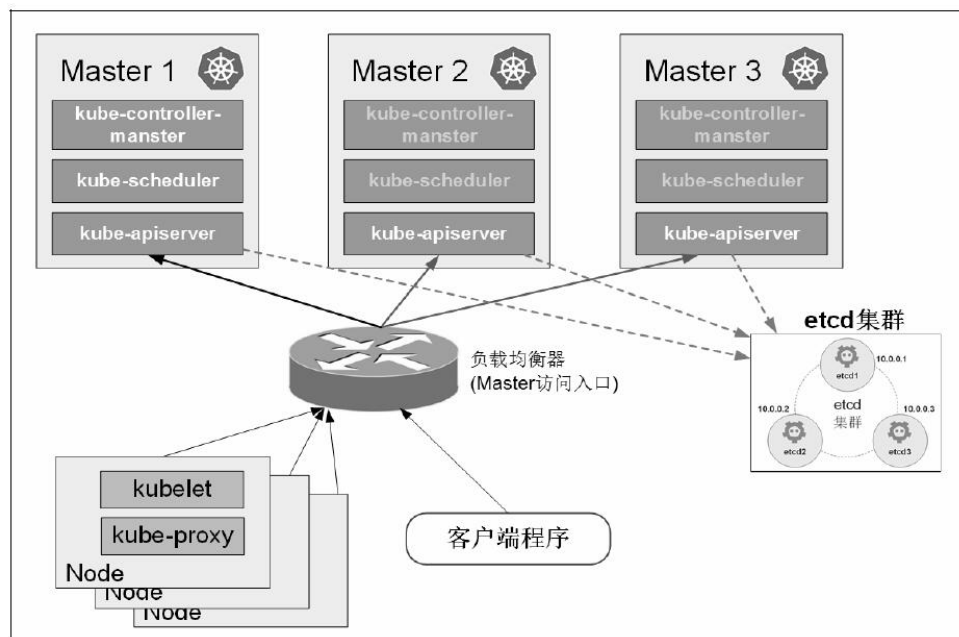


图5.6 KubernetesMaster高可用部署架构

Kubernetes建议Master的3个组件都以容器的形式启动，启动它们的基础工具是kubelet，所以它们都将以Static Pod的形式启动并由kubelet进行监控和自动重启。而kubelet本身的高可用则通过操作系统来完成，例如使用Linux的Systemd系统进行管理。

注意，如果之前已运行过这3个进程，则需要先停止它们，然后启动kubelet服务，这3个主进程将通过kubelet以容器的形式启动和运行。

接下来分别对 kube-apiserver 和 kube-controller-manager 、 kube-scheduler的高可用部署进行说明。

1) kube-apiserver的高可用部署

根据第2章的介绍，为kube-apiserver预先创建所有需要的CA证书和基本鉴权文件等内容，然后在每台服务器上创建其日志文件：

```
# touch /var/log/kube-apiserver.log
```

假设kubelet的启动参数指定--config=/etc/kubernetes/manifests，即Static Pod定义文件所在的目录，接下来就可以创建kube-apiserver.yaml配置文件用于启动kube-apiserver了。

```
kube-apiserver.yaml
apiVersion: v1
kind: Pod
metadata:
  name: kube-apiserver
spec:
```

```
    hostNetwork: true
    containers:
      - name: kube-apiserver
        image:    gcr.io/google_containers/kube-
apiserver:9680e782e08a1a1c94c656190011bd02
        command:
          - /bin/sh
          - -c
            - /usr/local/bin/kube-apiserver --etcd-
servers=http://127.0.0.1:2379
                                                    --admission-
control=NamespaceLifecycle,LimitRanger,SecurityContextDeny,
ServiceAccount,ResourceQuota
          --service-cluster-ip-range=169.169.0.0/16 --v=2
          --allow-privileged=False 1>>/var/log/kube-
apiserver.log 2>&1
        ports:
          - containerPort: 443
            hostPort: 443
            name: https
          - containerPort: 7080
            hostPort: 7080
            name: http
          - containerPort: 8080
            hostPort: 8080
            name: local
        volumeMounts:
```

- mountPath: /srv/kubernetes
name: srvkube
readOnly: true
- mountPath: /var/log/kube-apiserver.log
name: logfile
- mountPath: /etc/ssl
name: etcssl
readOnly: true
- mountPath: /usr/share/ssl
name: usrsharessl
readOnly: true
- mountPath: /var/ssl
name: varssl
readOnly: true
- mountPath: /usr/ssl
name: usrssl
readOnly: true
- mountPath: /usr/lib/ssl
name: usrlibssl
readOnly: true
- mountPath: /usr/local/openssl
name: usrlocalopenssl
readOnly: true
- mountPath: /etc/openssl
name: etcopenssl
readOnly: true
- mountPath: /etc/pki/tls

```
        name: etcpkits
        readOnly: true
volumes:
- hostPath:
    path: /srv/kubernetes
    name: srvkube
- hostPath:
    path: /var/log/kube-apiserver.log
    name: logfile
- hostPath:
    path: /etc/ssl
    name: etcssl
- hostPath:
    path: /usr/share/ssl
    name: usrsharessl
- hostPath:
    path: /var/ssl
    name: varssl
- hostPath:
    path: /usr/ssl
    name: usrssl
- hostPath:
    path: /usr/lib/ssl
    name: usrlibssl
- hostPath:
    path: /usr/local/openssl
    name: usrlocalopenssl
```

```
- hostPath:
    path: /etc/openssl
    name: etcopenssl
- hostPath:
    path: /etc/pki/tls
    name: etcpkits
```

其中，

- kube-apiserver需要使用hostNetwork模式，即直接使用宿主机网络，以使得客户端能够通过物理机访问其API。
- 镜像的tag来源于kubernetes发布包中的kube-apiserver.docker_tag文件：
kubernetes/server/kubernetes-server-linux-amd64/server/bin/kube-apiserver.docker_tag。
- --etcd-servers: 指定etcd服务的URL地址。
- 再加上其他必要的启动参数，包括--admission-control、--service-cluster-ip-range、CA证书相关配置等内容。
- 端口号的设置都配置了hostPort，将容器内的端口号直接映射为宿主机的端口号。

将 kube-apiserver.yaml 文件复制到 kubelet 监控的/etc/kubernetes/manifests目录下，kubelet将会自动创建yaml文件中定义的kube-apiserver的Pod。

接下来在另外两台服务器上重复该操作，使得每台服务器上启动一个kube-apiserver的Pod。

2) 为kube-apiserver配置负载均衡器

至此，我们启动了三个**kube-apiserver**实例，这三个**kube-apiserver**都可以正常工作，我们需要一个统一的、可靠的、允许部分**Master**节点故障的方式来访问它们，可以通过部署一个负载均衡器来实现。

在不同的平台下，负载均衡的实现方式不同：在一些公用云比如**GCE**、**AWS**、阿里云上都有现成的实现方案；对于本地集群，我们可以选择硬件或者软件来实现负载均衡，比如**Kubernetes**社区推荐的方案**haproxy**和**keepalived**来实现，其中**haproxy**做负载均衡，而**keepalived**负责对**haproxy**监控和进行高可用。

在完成**API Server**的负载均衡配置之后，对其访问还需要注意以下内容。

- 如果**Master**开启了安全认证机制，那么需要确保证书中包含负载均衡服务节点的IP。
- 对于外部的访问，比如通过**kubectl**访问**API Server**，那么需要配置为访问**API Server**对应的负载均衡器的IP地址。

3) **kube-controller-manager**和**kube-scheduler**的高可用配置

不同于**API Server**，**Master**中另外两个核心组件**kube-controller-manager**和**kube-scheduler**会修改集群的状态信息，因此对于**kube-controller-manager**和**kube-scheduler**而言，高可用不仅意味着需要启动多个实例，还需要这多个实例能实现选举并选举出**leader**，以保证同一时间只有一个实例可以对集群状态信息进行读写，避免出现同步问题和一致性问题。**Kubernetes**对于这种选举机制的实现是采用租赁锁（**lease-lock**）来实现的，我们可以通过在**kube-controller-manager**和

kube-scheduler的每个实例的启动参数中设置--leader-elect=true，来保证同一时间只会运行一个可修改集群信息的实例。

Scheduler和Controller Manager高可用的具体实现方式如下。

首先在每个Master节点上创建相应的日志文件：

```
# touch /var/log/kube-scheduler.log
# touch /var/log/kube-controller-manager.log
```

然后创建 kube-controller-manager 和 kube-scheduler 的 Pod 定义文件：

```
kube-controller-manager.yaml:
apiVersion: v1
kind: Pod
metadata:
  name: kube-controller-manager
spec:
  hostNetwork: true
  containers:
  - name: kube-controller-manager
    image: gcr.io/google_containers/kube-controller-
manager:fda24638d51a48baa13c35337fcd4793
    command:
    - /bin/sh
    - -c
      - /usr/local/bin/kube-controller-manager --
```

master=127.0.0.1:8080

--v=2 --leader-elect=true 1>>/var/log/kube-
controller-manager.log 2>&1

livenessProbe:

httpGet:

path: /healthz

port: 10252

initialDelaySeconds: 15

timeoutSeconds: 1

volumeMounts:

- mountPath: /srv/kubernetes
name: srvkube
readOnly: true
- mountPath: /var/log/kube-controller-manager.log
name: logfile
- mountPath: /etc/ssl
name: etcssl
readOnly: true
- mountPath: /usr/share/ssl
name: usrsharessl
readOnly: true
- mountPath: /var/ssl
name: varssl
readOnly: true
- mountPath: /usr/ssl
name: usrssl
readOnly: true

- mountPath: /usr/lib/ssl
name: usrlibssl
readOnly: true
- mountPath: /usr/local/openssl
name: usrlocalopenssl
readOnly: true
- mountPath: /etc/openssl
name: etcopenssl
readOnly: true
- mountPath: /etc/pki/tls
name: etcpkits
readOnly: true

volumes:

- hostPath:
path: /srv/kubernetes
name: srvkube
- hostPath:
path: /var/log/kube-controller-manager.log
name: logfile
- hostPath:
path: /etc/ssl
name: etcssl
- hostPath:
path: /usr/share/ssl
name: usrsharessl
- hostPath:
path: /var/ssl

```
    name: varssl
- hostPath:
    path: /usr/ssl
    name: usrssl
- hostPath:
    path: /usr/lib/ssl
    name: usrlibssl
- hostPath:
    path: /usr/local/openssl
    name: usrlocalopenssl
- hostPath:
    path: /etc/openssl
    name: etcopenssl
- hostPath:
    path: /etc/pki/tls
    name: etcpkits
```

其中,

- kube-controller-manager需要使用hostNetwork模式, 即直接使用宿主机网络。
- 镜像的tag来源于kubernetes发布包中的kube-controller-manager.docker_tag文件: kubernetes/server/kubernetes-server-linux-amd64/server/bin/kube-controller-manager.docker_tag。
- --master: 指定kube-apiserver服务的URL地址。
- --leader-elect=true: 使用leader选举机制。

```
kube-scheduler.yaml:
  apiVersion: v1
  kind: Pod
  metadata:
    name: kube-scheduler
  spec:
    hostNetwork: true
    containers:
      - name: kube-scheduler
        image: gcr.io/google_containers/kube-
scheduler:34d0b8f8b31e27937327961528739bc9
        command:
          - /bin/sh
          - -c
            - /usr/local/bin/kube-scheduler --
master=127.0.0.1:8080 --v=2 --leader-elect=true
1>>/var/log/kube-scheduler.log 2>&1
        livenessProbe:
          httpGet:
            path: /healthz
            port: 10251
          initialDelaySeconds: 15
          timeoutSeconds: 1
    volumeMounts:
      - mountPath: /var/log/kube-scheduler.log
        name: logfile
```

```

- mountPath:
  /var/run/secrets/kubernetes.io/serviceaccount
  name: default-token-s8ejd
  readOnly: true
volumes:
- hostPath:
  path: /var/log/kube-scheduler.log
  name: logfile

```

其中，

- kube-scheduler需要使用hostNetwork模式，即直接使用宿主机网络。
- 镜像的tag来源于kubernetes发布包中的kube-scheduler.docker_tag文件：
kubernetes/server/kubernetes-server-linux-amd64/server/bin/kube-scheduler.docker_tag。
- --master: 指定kube-apiserver服务的URL地址。
- --leader-elect=true: 使用leader选举机制。

将这两个yaml文件复制到kubelet监控的/etc/kubernetes/manifests目录下，kubelet将会自动创建yaml文件中定义的kube-controller-manager和kube-scheduler的Pod。

至此，我们完成了Kubernetes Master组件高可用的完整配置，配合etcd存储的高可用，整个Kubernetes集群的高可用已经全部完成。最后，只需要确认集群中所有访问API Server的地方都已经将访问地址修改为负载均衡的地址，就可以保证集群高可用的正常工作了。

3.Master高可用架构的演进

在当前的版本中， kubelet可以设置“--api-servers”启动参数来指定多个kube-apiserver，但是当第1个kube-apiserver不可用之后， kubelet无法连接到后面的kube-apiserver，也就是说只有第1个kube-apiserver起作用。如果这个问题得到解决，则kubelet无须通过额外的负载均衡器就能连接到多个API Server了。

另外，除了kubelet，其他核心组件kube-controller-manager、kube-scheduler和kube-proxy都需要配置kube-apiserver，目前它们的启动参数“--master”仅支持配置一个kube-apiserver，还无法支持多个kube-apiserver的配置。

Kubernetes计划在后续的版本中支持多个Master的配置，实现不需要负载均衡器的Master高可用架构。

5.1.6 Kubernetes集群监控

1.通过cAdvisor页面查看容器的运行状态

开源软件cAdvisor（Container Advisor）是用于监控容器运行状态的利器之一（cAdvisor项目的主页为<https://github.com/google/cadvisor>），它被用于多个与Docker相关的开源项目中。

在Kubernetes系统中，cAdvisor已被默认集成到了kubelet组件内，当kubelet服务启动时，它会自动启动cAdvisor服务，然后cAdvisor会实时采集所在节点的性能指标及在节点上运行的容器的性能指标。kubelet的启动参数--cadvisor-port可自定义cAdvisor对外提供服务的端口号，默认为4194。

cAdvisor提供了Web页面可供浏览器访问。例如Kubernetes集群中的一个Node的IP地址是192.168.18.3，则在浏览器中输入网址<http://192.168.18.3:4194>来访问cAdvisor的监控页面。cAdvisor的主页显示了主机的实时运行状态，包括CPU使用情况、内存使用情况、网络吞吐量及文件系统使用情况等信息。

图5.7展示了cAdvisor的几个性能监控页面。

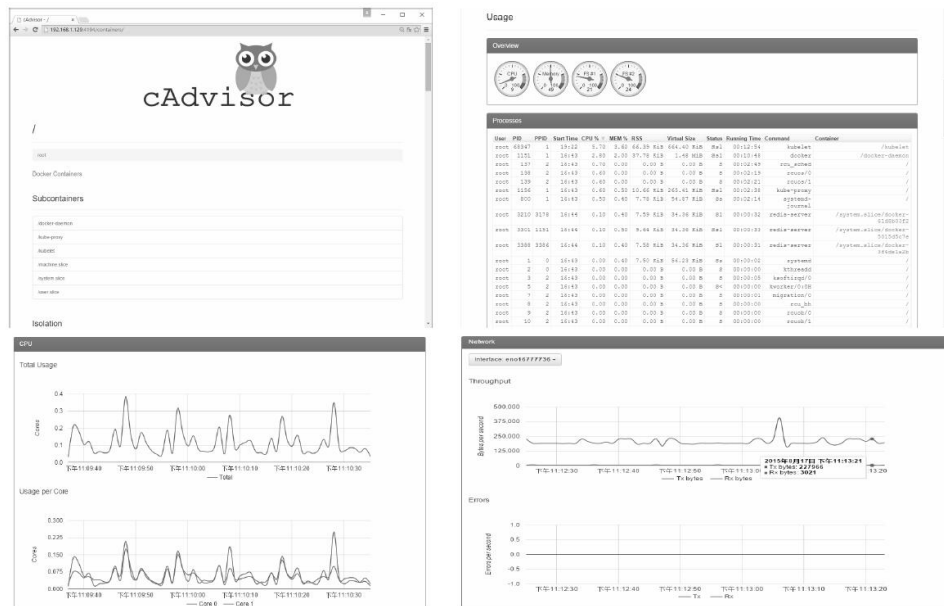


图5.7 主机的性能监控页面

通过Docker Containers链接可以查看容器列表及每个容器的性能数据，如图5.8所示。

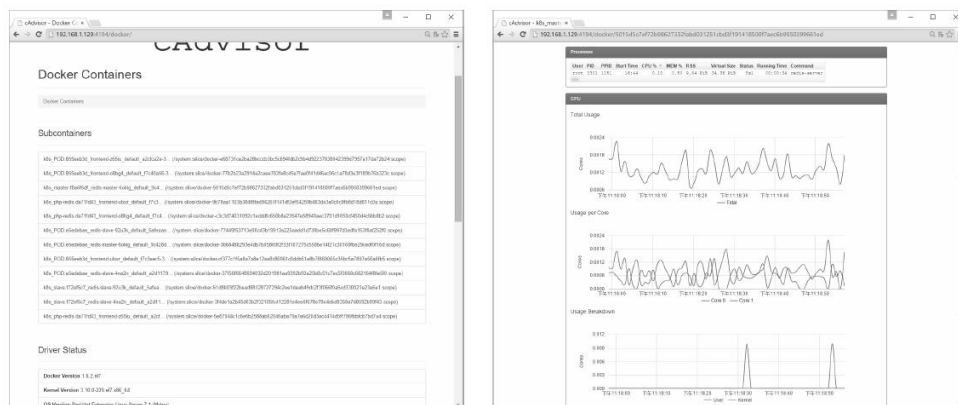


图5.8 容器的性能监控页面

此外，cAdvisor也提供了REST API供客户端远程调用，主要是为了定制开发，API返回的数据格式为JSON，可以采用如下URL来访

问:

```
http://<hostname>:<port>/api/<version>/<request>
```

例如，通过URL `http://192.168.18.3: 4194/api/v1.3/machine` 可以获取主机的相关信息:

```
{
  "num_cores":2,
  "cpu_frequency_khz":2793544,
  "memory_capacity":1915408384,
  "machine_id":"0f6233d8256a4ec1a673640e04b8344a",
  "system_uuid":"564D188F-8E82-21C0-6E89-
176E2C51EBB5",
  "boot_id":"a03d00d8-ca9c-4d74-a674-ebf5dfbc69d9",
  "filesystems":[
    {
      "device":"/dev/mapper/rhel-root",
      "capacity":18746441728
    },
    {
      "device":"/dev/sda1",
      "capacity":520794112
    }
  ],
  "disk_map":{
    "253:0":{
```

```
        "name": "dm-0",
        "major": 253,
        "minor": 0,
        "size": 2147483648,
        "scheduler": "none"
    },
    .....
},
"network_devices": [
    {
        "name": "eno16777736",
        "mac_address": "00:0c:29:51:eb:b5",
        "speed": 1000,
        "mtu": 1500
    }
],
"topology": [
    {
        "node_id": 0,
        "memory": 2146947072,
        "cores": [
            {
                "core_id": 0,
                "thread_ids": [
                    0
                ],
                "caches": null
            }
        ]
    }
]
```

```

        },
        .....
    ],
    "caches": [
        {
            "size": 6291456,
            "type": "Unified",
            "level": 3
        }
    ]
}
]
}

```

通过下面的URL则可以获得节点上最新（1分钟内）的容器的性能数据：
<http://192.168.1.129:4194/api/v1.3/subcontainers/system.slice/docker-5015d5c7ef72b98627332fabd031251cbd3f191418500f7aec6b9950399661ed.scope> °

结果为:

```

[
  {
    "name": "/system.slice/docker-5015d5c7ef72b98627332fabd031251cbd3f191418500f7aec6b9950399661ed.scope",

```

```
      "aliases": [
        "k8s_master.f8a6f6df_Redis-master-60kig_default_9c428d4f-4167-11e5-afe7-000c2921ba71_5dce2f85",
        "5015d5c7ef72b98627332fabd031251cbd3f191418500f7aec6b9950399661ed"
      ],
      "namespace": "docker",
      "spec": {
        "creation_time": "2015-08-17T08:44:27.401122502Z",
        "labels": {
          "io.kubernetes.pod.name": "default/Redis-master-60kig"
        },
        "has_cpu": true,
        "cpu": {
          "limit": 2,
          "max_limit": 0,
          "mask": "0-1"
        },
        "has_memory": true,
        "memory": {
          "limit": 18446744073709552000,
          "swap_limit": 18446744073709552000
        },
      },
```

```
        "has_network":true,
        "has_filesystem":false,
        "has_diskio":true
    },
    "stats":[
        {
            "timestamp":"2015-08-
18T00:54:26.167988505+08:00",
            "cpu":{
                "usage":{
                    "total":43121463207,
                    "per_cpu_usage":[
                        21578091763,
                        21543371444
                    ],
                    "user":4100000000,
                    "system":13620000000
                },
                "load_average":0
            },
            "diskio":{
                "io_service_bytes":[
                    {
                        "major":253,"minor":14,
                        "stats":{
                            "Async":8036352,"Read":8036352,"Sync":0,"Total":8036352,"Wr
```

ite":0

```
    }
  }
],
"io_serviced":[
  {
    "major":8,
    "minor":0,
    "stats":{
      "Async":0,
      .....
    ]
},
"memory":{
  "usage":16748544,
  "working_set":9297920,
  "container_data":{
    "pgfault":882,
    "pgmajfault":8
  },
  "hierarchical_data":{
    "pgfault":882,
    "pgmajfault":8
  }
},
"network":{
  "name":"","
```

```

    "rx_bytes":0,"rx_packets":0,"rx_errors":0,"rx_dropped":0,"t
x_bytes":0,"tx_packets":0,"tx_errors":0,"tx_dropped":0
    },
    "task_stats":{

    "nr_sleeping":0,"nr_running":0,"nr_stopped":0,"nr_uninterru
ptible":0,"nr_io_wait":0
    }
    },
    .....
    ]
    }
    ]

```

容器的性能数据对于集群监控非常有用，系统管理员可以根据cAdvisor提供的数据进行分析 and 告警。不过，由于cAdvisor是在每台Node上运行的，只能采集本机的性能指标数据，所以系统管理员需要对每台Node主机单独监控。

针对大型集群，Kubernetes建议使用几个开源软件组成的集成解决方案来实现对整个集群的监控。这些开源软件包括Heapster、InfluxDB及Grafana等。

2.Heapster+Influxdb+Grafana集群性能监控平台搭建

根据前面的说明，cAdvisor集成在kubelet中，运行在每个Node上，所以一个cAdvisor仅能对一台Node进行监控。在大规模容器集群中，需要对所有Node和全部容器进行性能监控，Kubernetes建议使用一套工具来实现集群性能数据的采集、存储和展示：Heapster、InfluxDB和Grafana。

- **Heapster**: 对集群中各Node上cAdvisor的数据采集汇聚的系统，通过访问每个Node上kubelet的API，再通过kubelet调用cAdvisor的API来采集该节点上所有容器的性能数据。Heapster对性能数据进行聚合，并将结果保存到后端存储系统中。Heapster支持多种后端存储系统，包括memory（保存在内存中）、InfluxDB、BigQuery、谷歌云平台提供的Google Cloud Monitoring（<https://cloud.google.com/monitoring/>）和Google Cloud Logging（<https://cloud.google.com/logging/>）等。Heapster项目的主页为<https://github.com/kubernetes/heapster>。
- **InfluxDB**: 是分布式时序数据库（每条记录都带有时间戳属性），主要用于实时数据采集、事件跟踪记录、存储时间图表、原始数据等。InfluxDB提供了REST API用于数据的存储和查询。InfluxDB的主页为<http://influxdb.com>。
- **Grafana**: 通过Dashboard将InfluxDB中的时序数据展现成图表或曲线等形式，便于运维人员查看集群的运行状态。Grafana的主页为<http://grafana.org>。

基于heapster+influxdb+grafana的集群监控系统总体架构如图5.9所示。

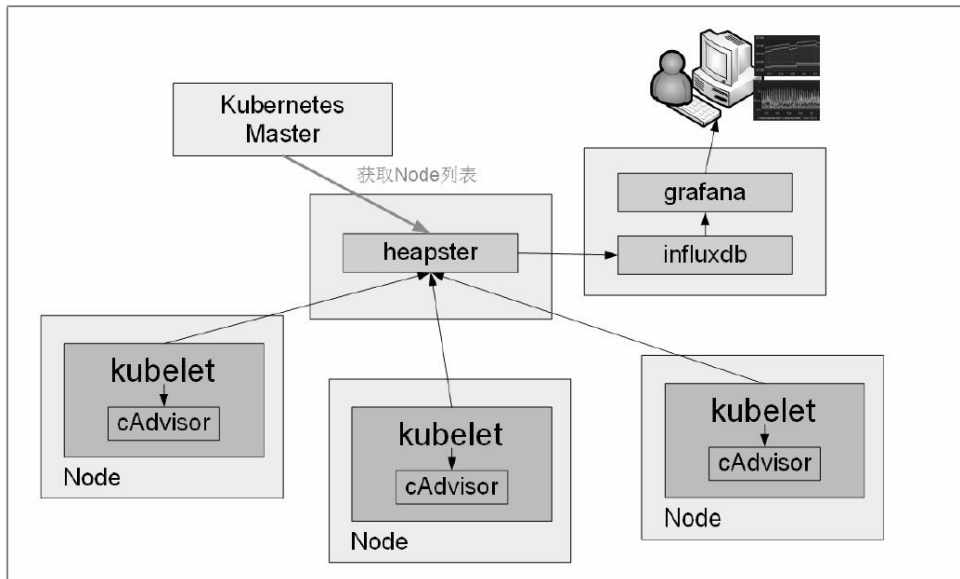


图5.9 Heapster集群监控系统架构图

Heapster、InfluxDB和Grafana均以Pod的形式启动和运行。由于Heapster需要与Kubernetes Master进行安全连接，所以需要设置Master的CA证书安全策略（参见第2章的说明）。

1) 部署Heapster、InfluxDB、Grafana容器应用

先创建它们的Service:

heapster-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  labels:
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: Heapster
```

```
name: heapster
namespace: kube-system
spec:
  ports:
    - port: 80
      targetPort: 8082
  selector:
    k8s-app: heapster
```

[influxdb-service.yaml](#)

```
apiVersion: v1
kind: Service
metadata:
  labels: null
  name: monitoring-InfluxDB
  namespace: kube-system
spec:
  type: NodePort
  ports:
    - name: http
      port: 8083
      targetPort: 8083
      nodePort: 8083
    - name: api
      port: 8086
      targetPort: 8086
```

```
nodePort: 8086
selector:
  name: influxGrafana
```

注意，这里使用type=NodePort将InfluxDB暴露在宿主机Node的端口上，以便我们使用浏览器对其进行访问。

grafana-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  labels:
    kubernetes.io/name: monitoring-Grafana
    kubernetes.io/cluster-service: "true"
  name: monitoring-Grafana
  namespace: kube-system
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 8085
  selector:
    name: influxGrafana
```

同样，使用`type=NodePort`将Grafana暴露在Node的端口上，以便客户端的浏览器对其进行访问。

使用`kubectl create`命令创建Services:

```
$ kubectl create -f heapster-service.yaml
$ kubectl create -f InfluxDB-service.yaml
$ kubectl create -f Grafana-service.yaml
```

在创建`heapster`容器之前，先创建InfluxDB和Grafana的RC，这两个容器将运行在同一个Pod中：

`influxdb-grafana-controller-v3.yaml`

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: monitoring-influxdb-grafana-v3
  namespace: kube-system
  labels:
    k8s-app: influxGrafana
    version: v3
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 1
  selector:
    k8s-app: influxGrafana
    version: v3
```

```

template:
  metadata:
    labels:
      k8s-app: influxGrafana
      version: v3
      kubernetes.io/cluster-service: "true"
  spec:
    containers:
      - image:
gcr.io/google_containers/heapster_influxdb:v0.5
        name: influxdb
        resources:
          # keep request = limit to keep this
container in guaranteed class
          limits:
            cpu: 100m
            memory: 500Mi
          requests:
            cpu: 100m
            memory: 500Mi
        ports:
          - containerPort: 8083
          - containerPort: 8086
        volumeMounts:
          - name: influxdb-persistent-storage
            mountPath: /data
      - image:

```

gcr.io/google_containers/heapster_grafana:v2.6.0-2

name: grafana

resources:

limits:

cpu: 100m

memory: 100Mi

requests:

cpu: 100m

memory: 100Mi

env:

This variable is required to setup
templates in Grafana.

- name: INFLUXDB_SERVICE_URL

value: http://monitoring-influxdb:8086

The following env variables are
required to make Grafana accessible via

the kubernetes api-server proxy. On
production clusters, we recommend

removing these env variables, setup
auth for grafana, and expose the grafana

service using a LoadBalancer or a
public IP.

- name: GF_AUTH_BASIC_ENABLED

value: "false"

- name: GF_AUTH_ANONYMOUS_ENABLED

value: "true"

- name: GF_AUTH_ANONYMOUS_ORG_ROLE

```
        value: Admin
      - name: GF_SERVER_ROOT_URL
        value: /api/v1/proxy/namespaces/kube-
system/services/monitoring-grafana/
    volumeMounts:
      - name: grafana-persistent-storage
        mountPath: /var
    volumes:
      - name: influxdb-persistent-storage
        emptyDir: {}
      - name: grafana-persistent-storage
        emptyDir: {}
```

注意，Grafana 容器环境变量 INFLUXDB_SERVICE_URL 设置为 InfluxDB 服务的所在地址。由于 Grafana 与 InfluxDB 处于同一个 Pod 中，所以 Grafana 使用 127.0.0.1 或 localhost 也可以访问到 InfluxDB 服务。

使用 `kubectl create` 命令创建该 RC：

```
$ kubectl create -f influxdb-grafana-controller-
v3.yaml
```

通过 `kubectl get pods--namespace=kube-system` 确认 Pod 成功启动：

```
# kubectl get pods --namespace=kube-system
```

	NAME	
STATUS	RESTARTS	AGE

READY

Running 0 4m

创建heapster容器，v1.1.0版本的heapster由4个容器组合为一个Pod:

heapster-controller-v1.1.0.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: heapster-v1.1.0
  namespace: kube-system
  labels:
    k8s-app: heapster
    kubernetes.io/cluster-service: "true"
    version: v1.1.0
spec:
  replicas: 1
  selector:
    matchLabels:
      k8s-app: heapster
      version: v1.1.0
  template:
    metadata:
      labels:
        k8s-app: heapster
```

```

        version: v1.1.0
spec:
  # 4 containers, 2 heapsters, 2 resizer
  containers:
    - image:
gcr.io/google_containers/heapster:v1.1.0
      name: heapster
      resources:
        # keep request = limit to keep this
container in guaranteed class
        limits:
          cpu: 100m
          memory: 200Mi
        requests:
          cpu: 100m
          memory: 200Mi
      command:
        - /heapster
    - --
source=kubernetes.summary_api:'192.168.18.3:8080'
    - --sink=influxdb:http://monitoring-
influxdb:8086
    - --metric_resolution=60s
    - image:
gcr.io/google_containers/heapster:v1.1.0
      name: eventer
      resources:

```

```
        # keep request = limit to keep this
container in guaranteed class
```

```
    limits:
```

```
        cpu: 100m
```

```
        memory: 200Mi
```

```
    requests:
```

```
        cpu: 100m
```

```
        memory: 200Mi
```

```
    command:
```

```
        - /eventer
```

```
        - --source=kubernetes:'192.168.18.3:8080'
```

```
            - --sink=influxdb:http://monitoring-
```

```
influxdb:8086
```

```
        - image: gcr.io/google_containers/addon-
```

```
resizer:1.3
```

```
name: heapster-nanny
```

```
resources:
```

```
    limits:
```

```
        cpu: 50m
```

```
        memory: 100Mi
```

```
    requests:
```

```
        cpu: 50m
```

```
        memory: 100Mi
```

```
env:
```

```
    - name: MY_POD_NAME
```

```
        valueFrom:
```

```
            fieldRef:
```

```
        fieldPath: metadata.name
-   name: MY_POD_NAMESPACE
    valueFrom:
        fieldRef:
            fieldPath: metadata.namespace
command:
-   /pod_nanny
-   --cpu=100m
-   --extra-cpu=0m
-   --memory=200Mi
-   --extra-memory=4Mi
-   --threshold=5
-   --deployment=heapster-v1.1.0
-   --container=heapster
-   --poll-period=300000
-   --estimator=exponential
-   image: gcr.io/google_containers/addon-
```

resizer:1.3

name: eventer-nanny

resources:

limits:

cpu: 50m

memory: 100Mi

requests:

cpu: 50m

memory: 100Mi

env:

```
- name: MY_POD_NAME
  valueFrom:
    fieldRef:
      fieldPath: metadata.name
- name: MY_POD_NAMESPACE
  valueFrom:
    fieldRef:
      fieldPath: metadata.namespace
command:
- /pod_nanny
- --cpu=100m
- --extra-cpu=0m
- --memory=200Mi
- --extra-memory=500Ki
- --threshold=5
- --deployment=heapster-v1.1.0
- --container=eventer
- --poll-period=300000
- --estimator=exponential
```

Heapster需要设置的启动参数如下。

(1) -source

配置采集来源，为Master URL地址：

```
--source=kubernetes.summary_api:'192.168.18.3:8080'
```

(2) -sink

配置后端存储系统，使用InfluxDB系统：

```
--sink=InfluxDB:http://monitoring-InfluxDB:8086
```

(3) --metric_resolution

性能指标的精度，60s表示将过去60秒的数据进行汇聚再进行存储。

其他参数可以通过进入heapster容器执行#heapster--help命令查看和设置。

注意，URL中的主机名地址使用的是InfluxDB的Service名字，这需要DNS服务正常工作，如果没有配置DNS服务，则也可以使用Service的ClusterIP地址。

值得说明的是，InfluxDB服务的名称没有加上命名空间，是因为Heapster服务与InfluxDB服务属于相同的命名空间kube-system。当然，使用带上命名空间的全服务名也是可以的，例如http://monitoring-influxdb.kube-system: 8086。

使用kubectl create命令完成创建该RC：

```
$ kubectl create -f heapster-controller-v1.1.0.yaml
```

通过kubectl get pods--namespace=kube-system确认Pod成功启动：

```
# kubectl get deployment --namespace=kube-system
```

	NAME	READY
STATUS	RESTARTS AGE	
	heapster-v1.1.0-1895667918-guisl	4/4
Running	0 3m	

查看heapster的日志，确保heapster成功在influxdb数据库中创建名为k8s的数据库：

```
# kubectl logs heapster-v1.1.0-1895667918-guisl -c
heapster --namespace=kube-system
```

```

I0706 09:36:15.313587      1 heapster.go:65]
/heapster --
source=kubernetes.summary_api:'192.168.18.3:8080' --
sink=influxdb:http://monitoring-influxdb:8086 --
metric_resolution=60s

I0706 09:36:15.313849      1 heapster.go:66] Heapster
version 1.1.0

I0706 09:36:15.314347      1 configs.go:60] Using
Kubernetes client with master "https://169.169.0.1:443" and
version "v1"

I0706 09:36:15.314371      1 configs.go:61] Using
kubelet port 10255

I0706 09:36:15.512107      1 influxdb.go:223] created
influxdb sink with options: host:monitoring-influxdb:8086
user:root db:k8s

I0706 09:36:15.512154      1 heapster.go:92] Starting

```

```
with InfluxDB Sink
    I0706 09:36:15.512163      1 heapster.go:92] Starting
with Metric Sink
    I0706 09:36:16.414060      1 heapster.go:171]
Starting heapster on port 8082
```

2) 查询InfluxDB数据库中的数据

让我们先通过InfluxDB的管理页面查看数据。

由于设置InfluxDB服务会暴露到物理Node节点上，所以我们可以通过任一Node的8083端口访问InfluxDB数据库提供的管理页面，如图5.10所示。通过右上角齿轮按钮可以修改连接属性（用于influxdb service设置为非默认端口号的时候）。单击右上角的Database下拉列表可以选择数据库，heapster创建的数据库名为k8s。

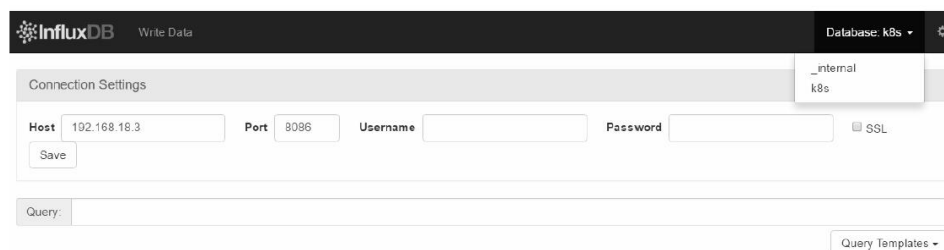


图5.10 InfluxDB管理页面

在Query输入框中输入“SHOW MEASUREMENTS”，即可查看所有measurements（序列表）。图5.11显示了部分measurements。

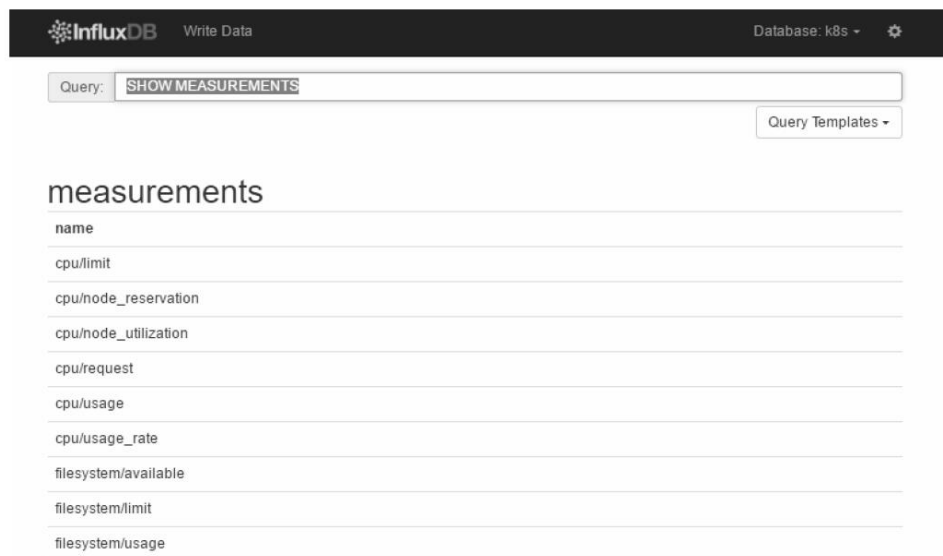


图5.11 show measurements结果页面

heapster采集的全部metric（性能指标）如表5.6所示。

表5.6 heapster采集的metric

metric 名称	说 明
cpu/limit	CPU hard limit，单位为毫秒
cpu/node_reservation	Node 保留的 CPU Share
cpu/node_utilization	Node 的 CPU 使用时间
cpu/request	CPU request，单位为毫秒
cpu/usage	全部 Core 的 CPU 累计使用时间
cpu/usage_rate	全部 Core 的 CPU 累计使用率，单位为毫秒
filesystem/usage	文件系统已用的空间，单位为字节
filesystem/limit	文件系统总空间限制，单位为字节
filesystem/available	文件系统可用的空间，单位为字节
memory/limit	Memory hard limit，单位为字节
memory/major_page_faults	major page faults 数量
memory/major_page_faults_rate	每秒的 major page faults 数量
memory/node_reservation	Node 保留的内存 Share
memory/node_utilization	Node 的内存使用值
memory/page_faults	page faults 数量
memory/page_faults_rate	每秒的 page faults 数量

续表

metric 名称	说 明
memory/request	Memory request, 单位为字节
memory/usage	总内存使用量
memory/working_set	总的 Working set usage, Working set 是指不会被 kernel 移除的内存
network/rx	累计接收的网络流量字节数
network/rx_errors	累计接收的网络流量错误数
network/rx_errors_rate	每秒接收的网络流量错误数
network/rx_rate	每秒接收的网络流量字节数
network/tx	累计发送的网络流量字节数
network/tx_errors	累计发送的网络流量错误数
network/tx_errors_rate	每秒发送的网络流量错误数
network/tx_rate	每秒发送的网络流量字节数
uptime	容器启动总时长

每个metric可以看作一张数据库表，表中每条记录由一组label组成，可以看作字段，如表5.7所示。

表5.7 metric的各label

Label 名称	说 明
pod_id	系统生成的 Pod 唯一名称
pod_name	用户指定的 Pod 名称
pod_namespace	Pod 所属的 namespace
container_base_image	容器的镜像名称
container_name	用户指定的容器名称
host_id	用户指定的 Node 主机名
hostname	容器运行所在主机名
labels	逗号分隔的 Label 列表
namespace_id	Pod 所属的 namespace 的 UID
resource_id	资源 ID

可以使用标准SQL SELECT语句对每个metric进行查询，例如查询CPU的使用时间：

```
select * from "cpu/usage" limit 10
```

结果如图5.12所示。

time	container_base_image	container_name	host_id	hostname	labels	namespace_id	namespace_name	node_name	pod_id	pod_name	pod_namespace	type	value
2015-08-08T21:32:02Z	gcr.io/google_containers/heapprod-v1.1.0	heapprod	k8s-node-1	k8s-node-1	k8s-app=heapprod-pod-template-hash=1895667918,version=v1.1.0	7f50e832-4450-11e6-b9fc-000c29ac2102	kube-system	k8s-node-1	31ef1664-5c16-11e6-bca7-000c29ac2102	heapprod-v1.1.0-1895667918-000c29ac2102	kube-system	pod_container	43049367
2015-08-08T21:32:02Z	gcr.io/google_containers/skydns-2015-10-13-8c728c	skydns	k8s-node-1	k8s-node-1	k8s-app=kube-dns,kubernetes.io/cluster=service-ipv4,version=v1.1	7f50e832-4450-11e6-b9fc-000c29ac2102	kube-system	k8s-node-1	ce7f0813-86d2-11e6-bca7-000c29ac2102	kube-dns-v1.1.0-000c29ac2102	kube-system	pod_container	234138702161
2015-08-08T21:32:02Z	gcr.io/google_containers/headlessflc-1.0	headlessflc	k8s-node-1	k8s-node-1	k8s-app=kube-dns,kubernetes.io/cluster=service-ipv4,version=v1.1	7f50e832-4450-11e6-b9fc-000c29ac2102	kube-system	k8s-node-1	ce7f0813-86d2-11e6-bca7-000c29ac2102	kube-dns-v1.1.0-000c29ac2102	kube-system	pod_container	203245018979
2015-08-08T21:32:02Z	gcr.io/google_containers/batman-ncscc-1.3	batman-ncscc	k8s-node-1	k8s-node-1	k8s-app=batman-pod-template-hash=1895667918,version=v1.1.0	7f50e832-4450-11e6-b9fc-000c29ac2102	kube-system	k8s-node-1	31ef1664-5c16-11e6-bca7-000c29ac2102	heapprod-v1.1.0-1895667918-000c29ac2102	kube-system	pod_container	27818862
2015-08-08T21:32:02Z			k8s-node-1	k8s-node-1				k8s-node-1				node	48632843818721
2015-08-08T21:32:02Z	gcr.io/google_containers/heapprod-v1.1.0	heapprod	k8s-node-1	k8s-node-1	k8s-app=heapprod-pod-template-hash=1895667918,version=v1.1.0	7f50e832-4450-11e6-b9fc-000c29ac2102	kube-system	k8s-node-1	31ef1664-5c16-11e6-bca7-000c29ac2102	heapprod-v1.1.0-1895667918-000c29ac2102	kube-system	pod_container	41655260
2015-08-08T21:32:02Z	gcr.io/google_containers/kube2sky-amd64-1.15	kube2sky	k8s-node-1	k8s-node-1	k8s-app=kube-dns,kubernetes.io/cluster=service-ipv4,version=v1.1	7f50e832-4450-11e6-b9fc-000c29ac2102	kube-system	k8s-node-1	ce7f0813-86d2-11e6-bca7-000c29ac2102	kube-dns-v1.1.0-000c29ac2102	kube-system	pod_container	17674965052
2015-08-08T21:32:02Z	gcr.io/google_containers/batman-ncscc-1.3	batman-ncscc	k8s-node-1	k8s-node-1	k8s-app=batman-pod-template-hash=1895667918,version=v1.1.0	7f50e832-4450-11e6-b9fc-000c29ac2102	kube-system	k8s-node-1	31ef1664-5c16-11e6-bca7-000c29ac2102	heapprod-v1.1.0-1895667918-000c29ac2102	kube-system	pod_container	41593308
2015-08-08T21:32:02Z	gcr.io/google_containers/heapprod-influxdb-v0.5	influxdb	k8s-node-1	k8s-node-1	k8s-app=influxdb,kubernetes.io/cluster=service-ipv4,version=v1.1	7f50e832-4450-11e6-b9fc-000c29ac2102	kube-system	k8s-node-1	1d56eac5-5c16-11e6-bca7-000c29ac2102	monitoring-influxdb-grafana-v3-000c29ac2102	kube-system	pod_container	427297277

图5.12 查询cpu/usage结果页面

3) Grafana页面查看和操作

访问Grafana服务需要通过Master代理模式进行访问，URL地址为 `http://192.168.18.3 : 8080/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana/`。

在grafana主页可以查看监控数据的图表展示画面。如图5.13所示为Cluster集群的整体信息，以折线图的形式展示了集群范围内各Node的CPU使用率、内存使用情况等信息。



图5.13 Grafana Cluster监控页面

图5.14显示的是所有Pod的信息，以折线图的形式展示了集群范围内各Pod的CPU使用率、内存使用情况、网络流量、文件系统使用情况等信息。

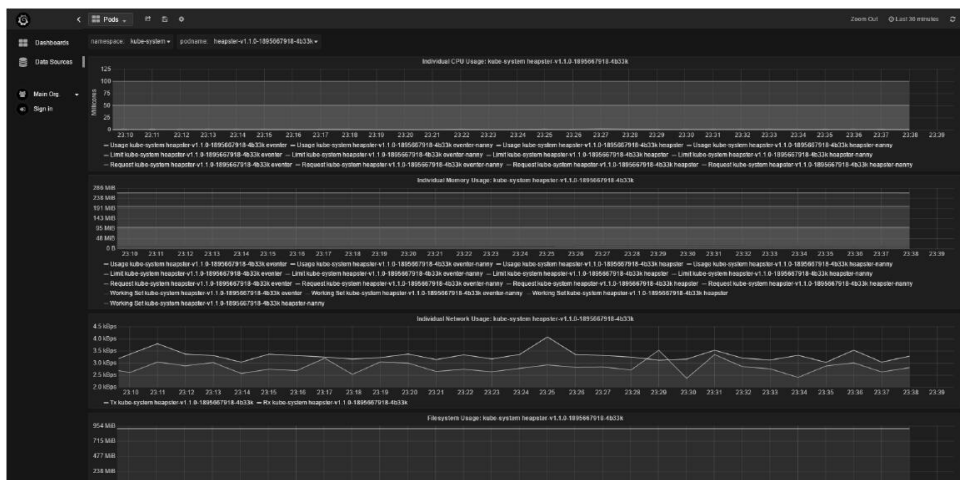


图5.14 Grafana Pod监控页面

Grafana页面上的每个图表都可以进行编辑，在标题上单击鼠标，点击“Edit”进入编辑页面，可以对每个metric进行个性化设置，例如查

询的表名、字段名、汇总计算等，如图5.15所示。

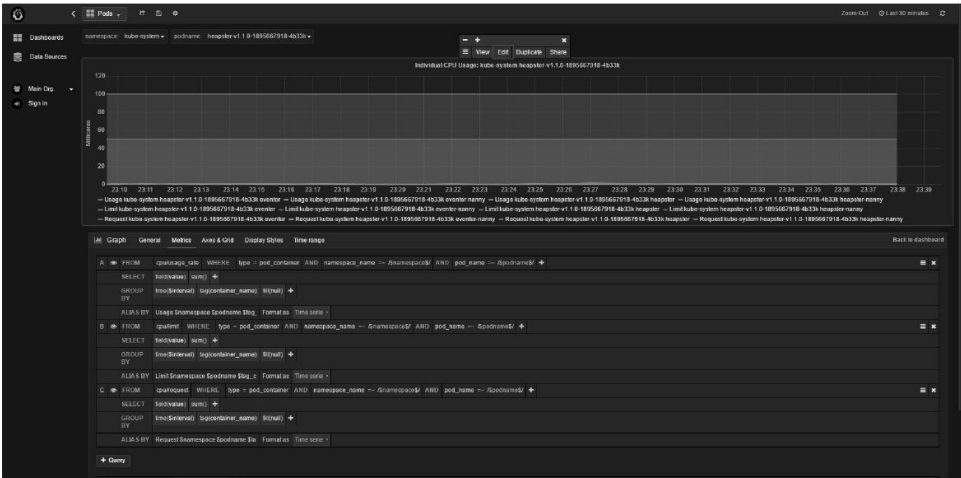


图5.15 编辑折线图

到此，基于heapster+influxdb+grafana的Kubernetes集群监控系统就搭建完成了。

5.1.7 kubelet的垃圾回收（GC）机制

Kubernetes集群中的垃圾回收（Garbage Collection，简称GC）机制由kubelet完成。kubelet定期清理不再使用的容器和镜像，每分钟进行一次容器GC操作，每5分钟进行一次镜像GC操作。

1.容器（Container）的GC设置

能够被GC清理的容器只能是仅由kubelet管理的容器。在kubelet所在的Node上直接通过docker run创建的容器将不会被kubelet进行GC清理操作。

kubelet的以下3个启动参数用于设置容器GC的条件。

- **--minimum-container-ttl-duration**: 已停止的容器在被清理之前的最小存活时间，例如“300ms”“10s”或“2h45m”，超过此存活时间的容器将被标记为可被GC清理，默认值为1分钟。
- **--maximum-dead-containers-per-container**: 以Pod为单位的可以保留的已停止的（属于同一Pod的）容器集的最大数量。有时，Pod中容器运行失败或者健康检查失败后，会被kubelet自动重启，这将产生一些停止的容器。默认值为2。
- **--maximum-dead-containers**: 在本Node上保留的已停止容器的最大数量，由于停止的容器也会消耗磁盘空间，所以超过该上限以后，kubelet会自动清理已停止的容器以释放磁盘空间，默认值为240。

如果需要关闭针对容器的GC操作，则可以将`--minimum-container-ttl-duration` 设置为 0，将 `--maximum-dead-containers-per-container` 和 `--maximum-dead-containers` 设置为负数。

2. 镜像（Image）的GC设置

Kubernetes系统中通过imageController和kublet中集成的cAdvisor共同管理镜像的生命周期，主要根据本Node的磁盘使用率来触发镜像的GC操作。

kubelet的以下3个启动参数用于设置镜像GC的条件。

- `--minimum-image-ttl-duration`: 不再使用的镜像在被清理之前的最小存活时间，例如“300ms”“10s”或“2h45m”，超过此存活时间的镜像被标记为可被GC清理，默认值为两分钟。
- `--image-gc-high-threshold`: 当磁盘使用率达到该值时，触发镜像的GC操作，默认值为90%。
- `--image-gc-low-threshold`: 当磁盘使用率降到该值时，GC操作结束，默认值为80%。

删除镜像的机制为：当磁盘使用率达到`image-gc-high-threshold`（例如90%）时触发，GC操作从最久未使用（Least Recently Used）的镜像开始删除，直到磁盘使用率降为`image-gc-low-threshold`（例如80%）或没有镜像可删为止。

5.2 Kubernetes高级案例

本节将对ElasticSearch日志管理平台的部署、Cassandra集群的部署及Kubernetes中容器的高级应用进行说明。

5.2.1 Elasticsearch日志搜集查询和展现案例

在Kubernetes集群环境中，一个完整的应用或服务都会涉及为数众多的组件运行，各组件所在的Node及实例数量都是可变的。日志子系统如果不做集中化管理，则会给系统的运维支撑造成很大的困难，因此有必要在集群层面对日志进行统一的收集和检索等工作。

容器中输出到控制台的日志，都会以*-json.log的命名方式保存在/var/lib/docker/containers/目录之下，这样就给了我们进行日志采集和后续处理的基础。

Kubernetes推荐采用Fluentd+ElasticSearch+Kibana完成对日志的采集、查询和展现工作。

在部署系统之前，需要以下两个前提条件。

- API Server正确配置了CA证书。
- DNS服务启动运行。

1.系统部署架构

系统的逻辑架构如图5.16所示。

在各Node上运行一个Fluentd容器，对本节点/var/log和/var/lib/docker/containers两个目录下的日志进程采集，然后汇总到ElasticSearch集群，最终通过Kibana完成和用户的交互工作。

这里有一个特殊的需求，Fluentd必须在每个Node上运行一份，为了满足这一需要，我们有以下几种不同的方式来部署Fluentd。

- 直接在Node主机上部署Fluentd。
- 利用kubelet的--config参数，为每个Node加载Fluentd Pod。
- 利用DaemonSet来让FluentdPod在每个Node上运行。

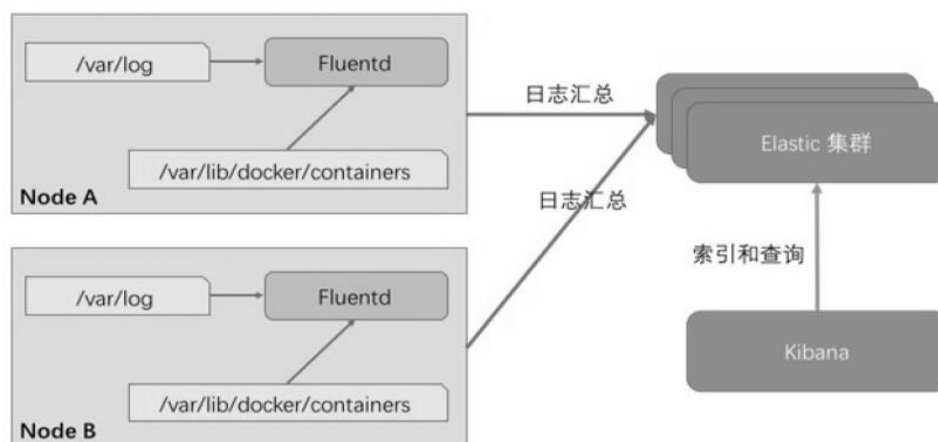


图5.16 Fluentd+ElasticSearch+Kibana系统逻辑架构图

目前官方推荐的包括Fluentd、Logstash等日志或者监控类的Pod的运行方式就是DaemonSet方式，因此本节我们也以这一方式进行配置。

2.创建ElasticSearch RC和服务

ElasticSearch的RC和服务定义：

```
elasticsearch-rc-svc.yml
---
apiVersion: v1
```

```

kind: ReplicationController
metadata:
  name: elasticsearch-logging-v1
  namespace: kube-system
  labels:
    k8s-app: elasticsearch-logging
    version: v1
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 2
  selector:
    k8s-app: elasticsearch-logging
    version: v1
  template:
    metadata:
      labels:
        k8s-app: elasticsearch-logging
        version: v1
        kubernetes.io/cluster-service: "true"
    spec:
      containers:
        - image:
gcr.io/google_containers/elasticsearch:1.8
          name: elasticsearch-logging
          resources:
            # keep request = limit to keep this
container in guaranteed class

```

```
      limits:
        cpu: 100m
      requests:
        cpu: 100m
    ports:
      - containerPort: 9200
        name: db
        protocol: TCP
      - containerPort: 9300
        name: transport
        protocol: TCP
    volumeMounts:
      - name: es-persistent-storage
        mountPath: /data
    volumes:
      - name: es-persistent-storage
        emptyDir: {}
  ---
  apiVersion: v1
  kind: Service
  metadata:
    name: elasticsearch-logging
    namespace: kube-system
  labels:
    k8s-app: elasticsearch-logging
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "Elasticsearch"
```

```
spec:
  ports:
    - port: 9200
      protocol: TCP
      targetPort: db
  selector:
    k8s-app: elasticsearch-logging
```

执行`kubectl create-f elastic-search.yml`命令完成创建。

命令成功执行后，首先验证Pod的运行情况。通过`kubectl get pods--namespaces=kube-system`获取运行中的Pod:

```
# kubectl get pods --namespaces=kube-system
```

		NAMESPACE	NAME
READY	STATUS	RESTARTS	AGE
		kube-system	elasticsearch-logging-v1-59qvp 1/1
Running	0	18h	
		kube-system	elasticsearch-logging-v1-xnv14 1/1
Running	0	18h	

接下来通过ElasticSearch的页面验证其功能。

执行`#kubectl cluster-info`命令获取ElasticSearch服务的地址:

```
# kubectl cluster-info
```

Elasticsearch is running at

```
http://192.168.18.3:8080/api/v1/proxy/namespaces/kube-  
system/services/elasticsearch-logging
```

接下来使用#`kubect1 proxy`命令对apiserver进行代理，成功执行后输出如下：

```
# kubect1 proxy  
Starting to serve on 127.0.0.1:8001
```

这样我们就可以在浏览器上访问URL地址`http://192.168.18.3:8001/api/v1/proxy/namespaces/kube-system/services/elasticsearch-logging`，来验证ElasticSearch的运行情况了，返回的内容是一个JSON文档：

```
{  
  "status": 200,  
  "name": "Emplate",  
  "cluster_name": "kubernetes-logging",  
  "version": {  
    "number": "1.5.2",  
    "build_hash":  
"62ff9868b4c8a0c45860bebb259e21980778ab1c",  
    "build_timestamp": "2015-04-27T09:21:06Z",  
    "build_snapshot": false,  
    "lucene_version": "4.10.4"  
  },  
}
```

```
"tagline": "You Know, for Search"
}
```

3.在每个Node上启动Fluentd

Fluentd的DaemonSet定义如下:

```
fluentd-ds.yml
---
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: fluentd-cloud-logging
  namespace: kube-system
  labels:
    k8s-app: fluentd-cloud-logging
spec:
  template:
    metadata:
      namespace: kube-system
      labels:
        k8s-app: fluentd-cloud-logging
    spec:
      containers:
        - name: fluentd-cloud-logging
          image: gcr.io/google_containers/fluentd-
elasticsearch:1.17
```

```
resources:
  limits:
    cpu: 100m
    memory: 200Mi
  env:
    - name: FLUENTD_ARGS
      value: -q
  volumeMounts:
    - name: varlog
      mountPath: /var/log
      readOnly: false
    - name: containers
      mountPath: /var/lib/docker/containers
      readOnly: false
  volumes:
    - name: containers
      hostPath:
        path: /var/lib/docker/containers
    - name: varlog
      hostPath:
        path: /var/log
```

通过kubect1 create命令创建Fluentd容器:

```
# kubect1 create -f fluentd-ds.yml
```

查看创建的结果:

```
# kubectl get daemonset
```

NAME	DESIRED	CURRENT	NODE- SELECTOR	AGE
fluentd-cloud-logging	3	3		<none> 1h

```
# kubectl get pods
```

STATUS	NAMESPACE	NAME	RESTARTS	AGE	READY
Running		fluentd-cloud-logging-7tw9z	0	18h	1/1
Running		fluentd-cloud-logging-aqdn1	0	18h	1/1
Running		fluentd-cloud-logging-o4usx	0	18h	1/1

结果显示Fluentd DaemonSet正常运行，启动3个Pod，与集群中的Node数量一致。

接下来，使用#kubectl logsfluentd-cloud-logging-7tw9z命令查看Pod的日志，在ElasticSearch正常工作的情况下，我们会看到类似下面这样的日志内容：

```
# kubectl logs fluentd-cloud-logging-7tw9z
```

```
Connection opened to Elasticsearch cluster =>
{:host=>"elasticsearch-logging", :port=>9200,
:scheme=>"http"}
```

说明Fluentd与ElasticSearch已经正确建立了连接。

4.运行Kibana

到此我们已经运行了ElasticSearch和Fluentd，数据的采集和汇聚过程已经完成，接下来就是使用Kibana来展示和操作数据了。

Kibana的RC和Service定义如下：

```
kibana-rc-svc.yml
---
apiVersion: v1
kind: ReplicationController
metadata:
  name: kibana-logging-v1
  namespace: kube-system
  labels:
    k8s-app: kibana-logging
    version: v1
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 1
  selector:
    k8s-app: kibana-logging
    version: v1
  template:
    metadata:
```

```
    labels:
      k8s-app: kibana-logging
      version: v1
      kubernetes.io/cluster-service: "true"
  spec:
    containers:
      - name: kibana-logging
        image: gcr.io/google_containers/kibana:1.3
        resources:
          # keep request = limit to keep this
container in guaranteed class
          limits:
            cpu: 100m
          requests:
            cpu: 100m
        env:
          - name: "ELASTICSEARCH_URL"
            value: "http://elasticsearch-logging:9200"
        ports:
          - containerPort: 5601
            name: ui
            protocol: TCP
    ---
  apiVersion: v1
  kind: Service
  metadata:
    name: kibana-logging
```

```
namespace: kube-system
labels:
  k8s-app: kibana-logging
  kubernetes.io/cluster-service: "true"
  kubernetes.io/name: "Kibana"
spec:
  ports:
    - port: 5601
      protocol: TCP
      targetPort: ui
  selector:
    k8s-app: kibana-logging
```

通过 `kubectl create-f kibana-rc-svc.yml` 命令创建 Kibana 的 RC 和 Service:

```
# kubectl create -f kibana-rc-svc.yml
replicationcontroller "kibana-logging-v1" created
service "kibana-logging" created
```

查看 Kibana 的运行情况:

```
# kubectl get pods
```

	NAMESPACE	NAME	READY
STATUS	RESTARTS	AGE	
	default	kibana-logging-v1-01akg	1/1
Running	0	1h	

```
# kubectl get svc
```

	NAME	CLUSTER-IP	EXTERNAL-IP
PORT(S)	AGE		
	kibana-logging	169.169.195.177	<none>
5601/TCP	1h		

```
# kubectl get rc
```

	NAME	DESIRED	CURRENT
AGE			
	kibana-logging-v1	1	1
			1h

结果表明运行均已成功。通过**kubectl cluster-info**命令获取Kibana服务的URL地址:

```
# kubectl cluster-info
```

Kibana is running at
<http://127.0.0.1:8080/api/v1/proxy/namespaces/kube-system/services/kibana-logging>

同样通过**kubectl proxy**命令启动代理，在出现**Starting to serve on 127.0.0.1: 8001**字样之后，用浏览器访问URL地址即可访问Kibana页面了：**http://192.168.18.3 : 8001/api/v1/proxy/namespaces/kube-system/services/kibana-logging**。

第1次进入页面需要进行一些设置，如图5.17所示，选择所需选项后单击**create**。

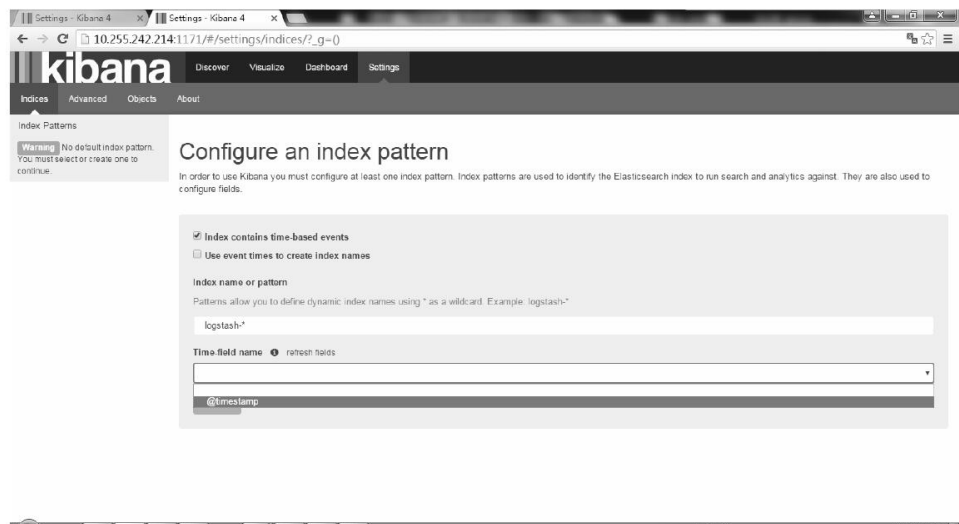


图5.17 Kibana创建索引页面

然后单击discover，就可以正常查询日志了，如图5.18所示。

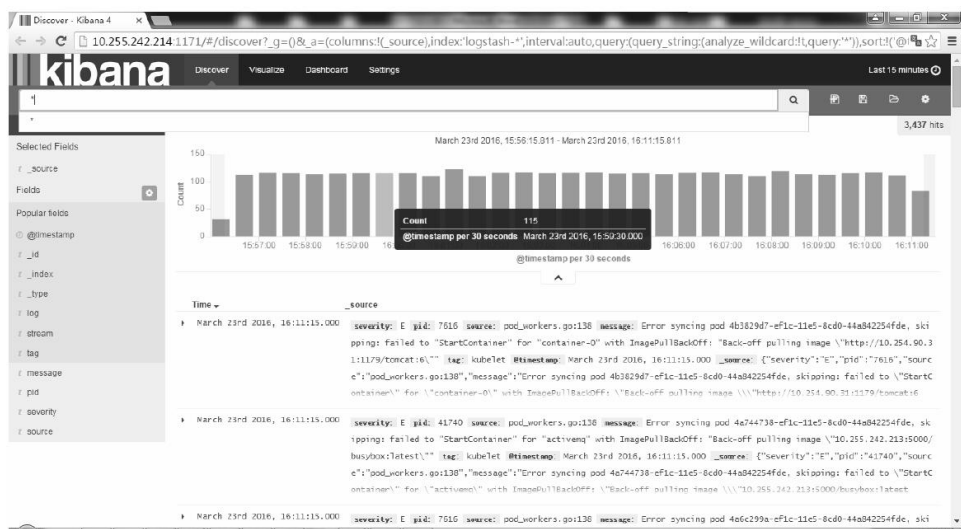


图5.18 Kibana查询日志页面

在搜索栏输入“error”关键字，可以搜索出从某些Node上找到的日志记录，如图5.19所示。



图5.19 Kibana日志关键字搜索页面

同时，通过左边菜单中Fields相关的内容对查询的内容进行限定，如图5.20所示。

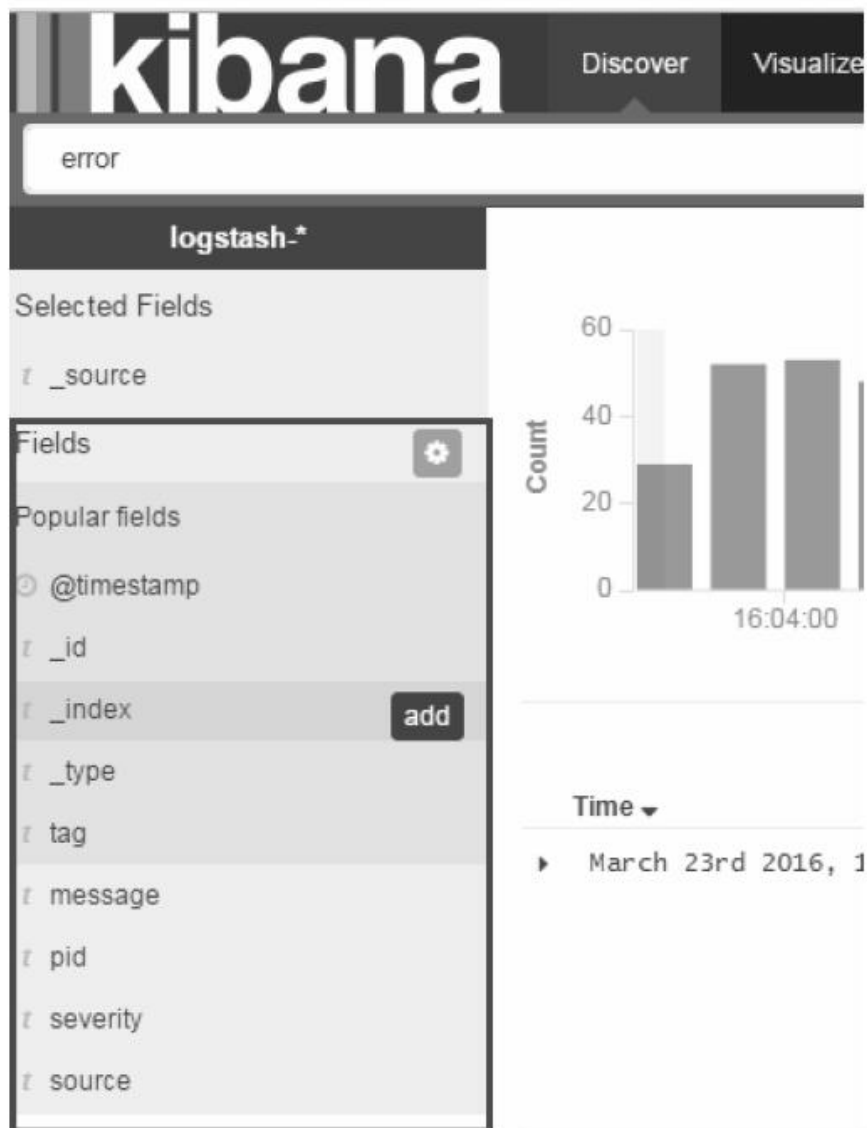


图5.20 Kibana查询日志

至此，Kubernetes集群范围内的统一日志收集和查询系统就搭建完成了。

5.2.2 Cassandra集群部署案例

Apache Cassandra是一套开源分布式NoSQL数据库系统，其主要特点就是它不是单个数据库，而是由一组数据库节点共同构成的一个分布式的集群数据库。由于Cassandra使用的是“去中心化”模式，所以当集群里的一个节点启动之后需要一个途径获知集群中新节点的加入。Cassandra使用了Seed（种子）的概念来完成在集群中节点之间的相互查找和通信。

本例通过对Kubernetes中Service概念的巧妙使用实现了各Cassandra节点之间的相互查找。

1. 自定义SeedProvider

在本例中使用了一个自定义的SeedProvider类来完成新节点的查询和添加，类名为io.k8s.cassandra.KubernetesSeedProvider。

KubernetesSeedProvider.java类的源代码节选如下：

```
.....

    public List<InetAddress> getSeeds() {
        List<InetAddress> list = new
ArrayList<InetAddress>();
        String host =
"https://kubernetes.default.cluster.local";
```

```

        String      serviceName      =
getEnvOrDefault("CASSANDRA_SERVICE", "cassandra");

        String      podNamespace      =
getEnvOrDefault("POD_NAMESPACE", "default");

        String      path      =
String.format("/api/v1/namespaces/%s/endpoints/",
podNamespace);

.....

    public static void main(String[] args) {
        SeedProvider provider = new
KubernetesSeedProvider(new HashMap<String, String>());
        System.out.println(provider.getSeeds());
    }
}

```

完整的源代码可以从这里获取：
<http://kubernetes.io/v1.0/examples/cassandra/java/src/io/k8s/cassandra/KubernetesSeedProvider.java>

创建Cassandra Pod的配置文件如下：

cassandra.yaml

```

apiVersion: v1
kind: Pod
metadata:
  labels:
    name: cassandra

```

```
    name: cassandra
spec:
  containers:
    - args:
      - /run.sh
      resources:
        limits:
          cpu: "0.5"
      image: gcr.io/google_containers/cassandra:v5
      name: cassandra
      ports:
        - name: cql
          containerPort: 9042
        - name: thrift
          containerPort: 9160
      volumeMounts:
        - name: data
          mountPath: /cassandra_data
      env:
        - name: MAX_HEAP_SIZE
          value: 512M
        - name: HEAP_NEWSIZE
          value: 100M
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
```

```
volumes:  
  - name: data  
    emptyDir: {}
```

需要说明的是，在镜像gcr.io/google_containers/cassandra: v5中安装了一个标准的Cassandra应用程序，并将定制的SeedProvider类——KubernetesSeedProvider打包到镜像中了。

定制的KubernetesSeedProvider类将使用REST API来访问Kubernetes Master，然后通过查询name=cassandra的服务指向的Pod来完成对其他“节点”的查找。

2.通过Service动态查找Pod

在KubernetesSeedProvider类中，通过查询环境变量CASSANDRA_SERVICE的值来获得服务的名称。这样就要求Service需要在Pod之前创建出来。如果我们已经创建好DNS服务（参见5.1节的案例介绍），那么也可以直接使用服务的名称而无须使用环境变量。

回顾一下Service的概念。Service通常用作一个负载均衡器，供Kubernetes集群中其他应用（Pod）对属于该Service的一组Pod进行访问。由于Pod的创建和销毁都会实时更新Service的Endpoints数据，所以可以动态地对Service的后端Pod进行查询了。Cassandra的“去中心化”设计使得Cassandra集群中的一个Cassandra实例（节点）只需要查询到其他节点，即可自动组成一个集群，正好可以使用Service的这个特性查询到新增的节点。图5.21描述了Cassandra新节点加入集群的过程。

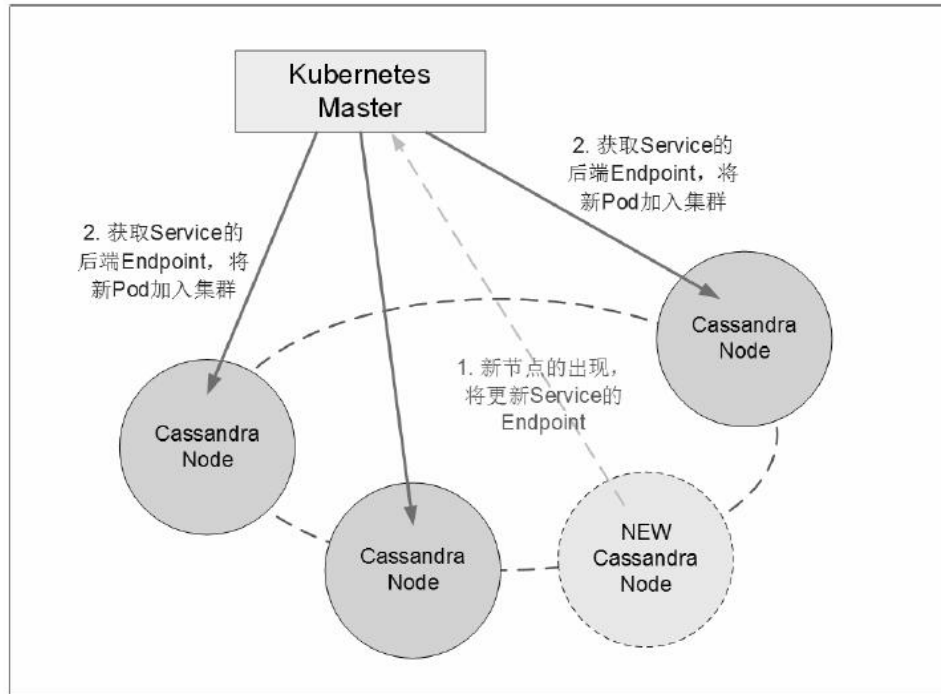


图5.21 Cassandra新节点加入集群的过程

在 Kubernetes 系统中，首先需要为 Cassandra 集群定义一个 Service。

cassandra-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: cassandra
  name: cassandra
spec:
  ports:
```

```
- port: 9042
selector:
  name: cassandra
```

在Service的定义中指定Label Selector为name=cassandra。

(1) 创建Service:

```
$ kubectl create -f cassandra-service.yaml
```

(2) 创建一个Cassandra Pod:

```
$ kubectl create -f cassandra-pod.yaml
```

现在，一个名为cassandra的Pod运行起来了，但还没有组成Cassandra集群。

(3) 创建一个RC来控制Pod集群:

cassandra-controller.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  labels:
    name: cassandra
  name: cassandra
spec:
```

```
replicas: 1
selector:
  name: cassandra
template:
  metadata:
    labels:
      name: cassandra
  spec:
    containers:
      - command:
          - /run.sh
        resources:
          limits:
            cpu: 0.5
        env:
          - name: MAX_HEAP_SIZE
            value: 512M
          - name: HEAP_NEWSIZE
            value: 100M
          - name: POD_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace
        image: gcr.io/google_containers/cassandra:v5
        name: cassandra
    ports:
      - containerPort: 9042
```

```
        name: cql
      - containerPort: 9160
        name: thrift
    volumeMounts:
      - mountPath: /cassandra_data
        name: data
  volumes:
    - name: data
      emptyDir: {}
```

由于在RC定义中指定的replicas数量为1，所以创建RC后，仍然只有之前创建的那个名为cassandra的Pod在运行。

3.Cassandra集群中新节点的自动添加

现在，我们使用Kubernetes提供的Scale（动态缩放）机制对Cassandra集群进行扩容：

```
$ kubectl scale rc cassandra --replicas=2
```

查看Pod，可以看到RC创建并启动了一个新的Pod：

```
$ kubectl get pods -l="name=cassandra"
```

NAME	READY	STATUS	RESTARTS	AGE
cassandra	1/1	Running	0	5m
cassandra-g52t3	1/1	Running	0	50s

使用Cassandra提供的nodetool工具对任一cassandra实例（Pod）进行访问来验证Cassandra集群的状态。下面的命令将访问名为cassandra的Pod（访问cassandra-g52t3也能获得相同的结果）：

```
$ kubectl exec -ti cassandra -- nodetool status
```

Datacenter: datacenter1

=====

Status=Up/Down

|/ State=Normal/Leaving/Joining/Moving

--	Address	Load	Tokens	Owns (effective)
Host ID		Rack		
	UN 10.1.20.16	51.58 KB	256	100.0%
1625c65d-b5b6-40f4-a794-6f5a12322d86		rack1		
	UN 10.1.10.11	51.51 KB	256	100.0%
cdfcbf1a-795c-4412-9d3f-e8fe50bb8deb		rack1		

可以看到Cassandra集群中有两个节点处于正常运行状态（Up and Normal, UN）。结果中的两个IP地址为两个Cassandra Pod的IP地址。

内部的过程为：每个Cassandra节点（Pod）通过API访问Kubernetes Master，查询名为cassandra的Service的Endpoints（即Cassandra节点），若发现有新节点加入，就进行添加操作，最后成功组成了一个Cassandra集群。

我们再增加两个Cassandra实例：

```
$ kubectl scale rc cassandra --replicas=4
```

用nodetool工具查看Cassandra集群状态:

```
$ kubectl exec -ti cassandra -- nodetool status

Datacenter: datacenter1
=====

Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address            Load            Tokens   Owns (effective)
Host ID                Rack
UN    10.1.20.16          51.58 KB        256      50.5%
1625c65d-b5b6-40f4-a794-6f5a12322d86 rack1
UN    10.1.10.12          52.03 KB        256      47.0%
8bcc1c3e-44ec-46a7-b981-4090b206f14e rack1
UN    10.1.20.17          68.05 KB        256      50.6%
579b6493-e92a-47f5-91f2-9313198a24c9 rack1
UN    10.1.10.11          51.51 KB        256      51.9%
cdfcbf1a-795c-4412-9d3f-e8fe50bb8deb rack1
```

可以看到4个Cassandra节点都加入Cassandra集群中了。

另外，可以通过查看Cassandra Pod的日志来看到新节点加入集群的记录:

```
$ kubectl logs cassandra-g52t3

.....

INFO 18:05:36 Handshaking version with /10.1.20.17
INFO 18:05:36 Node /10.1.20.17 is now part of the
```

cluster

INFO 18:05:36 InetAddress /10.1.20.17 is now UP

INFO 18:05:38 Handshaking version with /10.1.10.12

INFO 18:05:39 Node /10.1.10.12 is now part of the

cluster

INFO 18:05:39 InetAddress /10.1.10.12 is now UP

本例描述了一种通过API查询Service来完成动态Pod发现的应用场景。对于类似于Cassandra集群的应用，都可以使用对Service进行查询后端Endpoints这种巧妙的方法来实现对应用集群（属于同一Service）中新加入节点的查找。

5.3 Trouble Shooting指导

本节将对Kubernetes集群中常见的问题的排查方法进行说明。

为了跟踪和发现Kubernetes集群中运行的容器应用出现的问题，常用的查错方法如下。

首先，查看Kubernetes对象的当前运行时信息，特别是与对象关联的Event事件。这些事件记录了相关主题、发生时间、最近发生时间、发生次数及事件原因等，对排查故障非常有价值。此外，通过查看对象的运行时数据，我们还可以发现参数错误、关联错误、状态异常等明显问题。由于Kubernetes中多种对象相互关联，因此，这一步可能会涉及多个相关对象的排查问题。

其次，对于服务、容器的问题，则可能需要深入容器内部进行故障诊断，此时可以通过查看容器的运行日志来定位具体问题。

最后，对于某些复杂问题，比如Pod调度这种全局性的问题，可能需要结合集群中每个节点上的Kubernetes服务日志来排查。比如搜集Master上kube-apiserver、kube-schedule、kube-controler-manager服务的日志，以及各个Node节点上的kubelet、kube-proxy服务的日志，综合判断各种信息，我们就能找到问题的原因并解决问题。

5.3.1 查看系统Event事件

在Kubernetes集群中创建了Pod之后，我们可以通过`kubectl get pods`命令查看Pod列表，但该命令能够显示的信息很有限。Kubernetes提供了`kubectl describe pod`命令来查看一个Pod的详细信息。

```
$ kubectl describe pod redis-master-bobr0
Name:                                Redis-master-bobr0
Namespace:                           default
Image(s):                            kubeguide/Redis-master
Node:                                k8s-node-1/192.168.18.3
Labels:                               name=Redis-master,role=master
Status:                               Running
Reason:
Message:
IP:                                   172.17.0.58
Replication Controllers:              Redis-master (1/1 replicas created)
Containers:
  master:
    Image:          kubeguide/Redis-master
    Limits:
```

cpu: 250m
memory: 64Mi
State: Running
Started: Fri, 21 Aug 2015 14:45:37
+0800

Ready: True
Restart Count: 0

Conditions:

Type	Status
Ready	True

Events:

FirstSeen	LastSeen	Count
From	SubobjectPath	Reason
Message		
Fri, 21 Aug 2015 14:45:36 +0800	Fri, 21 Aug 2015 14:45:36 +0800	1
{kubelet	k8s-node-1}	
implicitly required container	Pod	
container	image	
"myregistry:5000/google_containers/pause:latest"	already	
present on machine		
Fri, 21 Aug 2015 14:45:37 +0800	Fri, 21 Aug 2015 14:45:37 +0800	1
{kubelet	k8s-node-1}	
implicitly required container	Pod	
created	Created	
with docker id a4aa97813908		
Fri, 21 Aug 2015 14:45:37 +0800	Fri, 21 Aug 2015 14:45:37 +0800	1
{kubelet	k8s-node-1}	
implicitly required container	Pod	
started	Started	
with docker id a4aa97813908		

```

Fri, 21 Aug 2015 14:45:37 +0800      Fri, 21 Aug
2015 14:45:37 +0800 1                  {kubelet k8s-node-1}
spec.containers{master}                created
Created with docker id 1e746245f768

Fri, 21 Aug 2015 14:45:37 +0800      Fri, 21 Aug
2015 14:45:37 +0800 1                  {kubelet k8s-node-1}
spec.containers{master}                started
Started with docker id 1e746245f768

Fri, 21 Aug 2015 14:45:37 +0800      Fri, 21 Aug
2015 14:45:37 +0800 1                  {scheduler }
scheduled                               Successfully assigned Redis-master-bobr0
to k8s-node-1

```

该命令除了显示Pod创建时的配置定义、状态等信息，还显示了与该Pod相关的最近的Event事件，事件信息对于查错非常有用。如果某个Pod一直处于Pending状态，则我们通过kubect describe命令就能了解到失败的具体原因。例如，从Event事件中我们可能获知Pod失败的原因有以下几种。

- 没有可用的Node以供调度。
- 开启了资源配额管理并且当前Pod的目标节点上恰好没有可用的资源。
- 正在下载镜像。

kubect describe 命令还可用于查看其他 Kubernetes 对象，包括 Node、RC、Service、Namespace、Secrets等，对于每一种对象都会显示相关联的其他信息。

例如，查看一个服务的详细信息：

```
$ kubectl describe service redis-master

Name:                Redis-master
Namespace:            default
Labels:               name=Redis-master
Selector:             name=Redis-master
Type:                 ClusterIP
IP:                   169.169.208.57
Port:                 <unnamed>      6379/TCP
Endpoints:            172.17.0.58:6379
Session Affinity:     None
No events.
```

如果查看的对象属于某个特定的namespace，则需要加上--namespace=<namespace>进行查询。例如：

```
$ kubectl get service kube-dns --namespace=kube-system
```

5.3.2 查看容器日志

在需要排查容器内部应用程序生成的日志时，我们可以使用 `kubectl logs<pod_name>` 命令：

```
$ kubectl logs redis-master-bobr0

[1] 21 Aug 06:45:37.781 * Redis 2.8.19 (00000000/0) 64
bit, stand alone mode, port 6379, pid 1 ready to start.

[1] 21 Aug 06:45:37.781 # Server started, Redis
version 2.8.19

[1] 21 Aug 06:45:37.781 # WARNING overcommit_memory is
set to 0! Background save may fail under low memory
condition. To fix this issue add 'vm.overcommit_memory = 1'
to /etc/sysctl.conf and then reboot or run the command
'sysctl vm.overcommit_memory=1' for this to take effect.

[1] 21 Aug 06:45:37.782 # WARNING you have Transparent
Huge Pages (THP) support enabled in your kernel. This will
create latency and memory usage issues with Redis. To fix
this issue run the command 'echo never >
/sys/kernel/mm/transparent_hugepage/enabled' as root, and
add it to your /etc/rc.local in order to retain the setting
after a reboot. Redis must be restarted after THP is
disabled.

[1] 21 Aug 06:45:37.782 # WARNING: The TCP backlog
```

```
setting of 511 cannot be enforced because  
/proc/sys/net/core/somaxconn is set to the lower value of  
128.
```

如果在一个Pod中包含多个容器，则需要通过-c参数指定容器的名称来进行查看，例如：

```
kubectl logs <pod_name> -c <container_name>
```

这个命令与在Pod的宿主机上运行docker logs<container_id>的效果是一样的。

容器中应用程序生成的日志与容器的生命周期是一致的，所以在容器被销毁之后，容器内部的文件也会被丢弃，包括日志等。如果需要保留容器内应用程序生成的日志，则一方面可以使用挂载的Volume（存储卷）将容器产生的日志保存到宿主机，另一方面也可以通过一些工具对日志进行采集，包括Fluentd、ElasticSearch等开源软件。

5.3.3 查看Kubernetes服务日志

如果在Linux系统上进行安装，并且使用systemd系统来管理Kubernetes服务，那么systemd的journal系统会接管服务程序的输出日志。在这种环境中，可以通过使用systemd status或journalctl工具来查看系统服务的日志。

例如，使用systemctl status命令查看kube-controller-manager服务的日志：

```
# systemctl status kube-controller-manager -l
kube-controller-manager.service - Kubernetes
Controller Manager
    Loaded: loaded (/usr/lib/systemd/system/kube-
controller-manager.service; enabled)
    Active: active (running) since Fri 2015-08-21
18:36:29 CST; 5min ago
Docs:
https://github.com/GoogleCloudPlatform/kubernetes
Main PID: 20339 (kube-controller)
    CGroup: /system.slice/kube-controller-
manager.service
           └─20339 /usr/bin/kube-controller-manager --
logtostderr=false --v=4 --master=http://kubernetes-
master:8080 --log_dir=/var/log/kubernetes
```

```
Aug 21 18:36:29 kubernetes-master systemd[1]: Starting  
Kubernetes Controller Manager...
```

```
Aug 21 18:36:29 kubernetes-master systemd[1]: Started  
Kubernetes Controller Manager.
```

使用journalctl命令查看:

```
# journalctl -u kube-controller-manager  
-- Logs begin at Mon 2015-08-17 16:43:22 CST, end at  
Fri 2015-08-21 18:36:29 CST. --
```

```
Aug 17 16:44:14 kubernetes-master systemd[1]: Starting  
Kubernetes Controller Manager...
```

```
Aug 17 16:44:14 kubernetes-master systemd[1]: Started  
Kubernetes Controller Manager.
```

如果不使用systemd系统接管Kubernetes服务的标准输出，则也可以通过日志相关的启动参数来指定日志的存放目录。

- `--logtostderr=false`: 不输出到stderr。
- `--log-dir=/var/log/kubernetes`: 日志的存放目录。
- `--alsologtostderr=false`: 设置为true则表示将日志输出到文件时也输出到stderr。
- `--v=0`: glog日志级别。
- `--vmodule=gfs*=2, test*=4`: glog基于模块的详细日志级别。

在`--log_dir`设置的目录中可以查看各服务进程生成的日志文件，日志文件的数量和大小依赖于日志级别的设置。例如kube-controller-

manager可能生成的几个日志文件如下。

- kube-controller-manager.ERROR。
- kube-controller-manager.INFO。
- kube-controller-manager.WARNING。
- kube-controller-manager.kubernetes-master.unknownuser.log.ERROR.20150930-173939.9847。
- kube-controller-manager.kubernetes-master.unknownuser.log.INFO.20150930-173939.9847。
- kube-controller-manager.kubernetes-master.unknownuser.log.WARNING.20150930-173939.9847。

在大多数情况下，我们从WARNING和ERROR级别的日志中就能找到问题的原因，但有时还是需要排查INFO级别的日志甚至DEBUG级别的详细日志。此外，etcd服务也属于Kubernetes集群中的重要组成部分，所以它的日志也不能忽略。

如果是某个Kubernetes对象存在问题，则我们可以用这个对象的名字作为关键字搜索Kubernetes的日志来发现和解决问题。在大多数情况下，我们平常所遇到的主要是与Pod对象相关的问题，比如无法创建Pod、Pod启动后就停止或者Pod副本无法增加等。此时，我们可以先确定Pod在哪个节点上，然后登录这个节点，从kubelet的日志中查询该Pod的完整日志，然后进行问题排查。对于与Pod扩容相关或者与RC相关的问题，则很可能在kube-controller-manager及kube-scheduler的日志上找出问题的关键点。

另外，kube-proxy经常被我们忽视，因为即使它意外地被停止，Pod的状态也是正常的，但会导致某些服务访问异常的情况。这些错

误通常与每个节点上的**kube-proxy**服务有着密切的关系。遇到这些问题时，首先要排查**kube-proxy**服务的日志，同时排查防火墙服务，特别是要留意防火墙中是否有人为添加的可疑规则。

5.3.4 常见问题

本节对Kubernetes系统中的一些常见问题及解决方法进行说明。

1.由于无法下载pause镜像导致Pod一直处于Pending的状态

以redis-master为例，使用如下配置文件redis-master-controller.yaml创建RC和Pod:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-master
  labels:
    name: redis-master
spec:
  replicas: 1
  selector:
    name: redis-master
  template:
    metadata:
      labels:
        name: redis-master
    spec:
```

```
containers:
- name: master
  image: kubeguide/redis-master
  ports:
  - containerPort: 6379
```

执行`kubectl create-f redis-master-controller.yaml`成功。

但在查看Pod时，发现其总是无法处于Running状态。通过`kubectl get pods`命令可以看到：

```
$ kubectl get pods
```

	NAME	READY	STATUS
RESTARTS	AGE		
	redis-master-6yy7o	0/1	Image: kubeguide/redis-master is ready, container is creating
		0	5m

进一步使用`kubectl describe pod redis-master-6yy7o`命令查看该Pod的详细信息：

```
$ kubectl describe pod redis-master-6yy7o
```

Name:	redis-master-6yy7o
Namespace:	default
Image(s):	kubeguide/redis-master
Node:	127.0.0.1/127.0.0.1
Labels:	name=redis-master
Status:	Pending

Reason:

Message:

IP:

Replication Controllers: redis-master (1/1
replicas created)

Containers:

master:

Image: kubeguide/redis-master

State: Waiting

Reason: Image: kubeguide/redis-master
is ready, container is creating

Ready: False

Restart Count: 0

Conditions:

Type	Status
------	--------

Ready	False
-------	-------

Events:

FirstSeen	LastSeen	Count
From	SubobjectPath	Reason
Thu, 24 Sep 2015 19:19:25 +0800	Thu, 24 Sep 2015 19:25:58 +0800	3
{kubelet 127.0.0.1} failedSync Error syncing pod, skipping: image pull failed for gcr.io/google_containers/pause-amd64:3.0, this may be because there are no credentials on this request. details: (API error (500): invalid registry endpoint https://gcr.io/v0/: unable to ping registry endpoint https://gcr.io/v0/v2 ping attempt failed with error: Get		

https://gcr.io/v2/: dial tcp 173.194.196.82:443: connection refused v1 ping attempt failed with error: Get https://gcr.io/v1/_ping: dial tcp 173.194.79.82:443: connection refused. If this private registry supports only HTTP or HTTPS with an unknown CA certificate, please add `--insecure-registry gcr.io` to the daemon's arguments. In the case of HTTPS, if you have access to the registry's CA certificate, no need for the flag; simply place the CA certificate at /etc/docker/certs.d/gcr.io/ca.crt)

Thu, 24 Sep 2015 19:19:25 +0800 Thu, 24 Sep 2015 19:25:58 +0800 3 {kubelet 127.0.0.1} implicitly required container POD failed Failed to pull image "gcr.io/google_containers/pause-amd64:3.0": image pull failed for gcr.io/google_containers/pause:0.8.0, this may be because there are no credentials on this request. details: (API error (500): invalid registry endpoint https://gcr.io/v0/: unable to ping registry endpoint https://gcr.io/v0/v2 ping attempt failed with error: Get https://gcr.io/v2/: dial tcp 173.194.196.82:443: connection refused v1 ping attempt failed with error: Get https://gcr.io/v1/_ping: dial tcp 173.194.79.82:443: connection refused. If this private registry supports only HTTP or HTTPS with an unknown CA certificate, please add `--insecure-registry gcr.io` to the daemon's arguments. In the case of HTTPS, if you have access to the registry's CA certificate, no need for the flag; simply place the CA certificate at /etc/docker/certs.d/gcr.io/ca.crt

可以看到，该Pod的状态为Pending，从Message部分显示的信息可以看出其原因是image pull failed for gcr.io/google_containers/pause-amd64: 3.0，说明系统在创建Pod时无法从gcr.io下载pause镜像，所以导致创建Pod失败。

解决方法如下。

(1) 如果服务器可以访问Internet，并且不希望使用HTTPS的安全机制来访问gcr.io，则可以在Docker Daemon的启动参数中加上--insecure-registry gcr.io来表示可以进行匿名下载。

(2) 如果Kubernetes集群环境在内网环境中，无法访问gcr.io网站，则可以先通过一台能够访问gcr.io的机器将pause镜像下载下来，导出后，再导入内网的Docker私有镜像库中，并在kubelet的启动参数中加上--pod_infra_container_image，配置为：

```
--pod_infra_container_image=<docker_registry_ip>:  
<port>/google_containers/pause-amd64:3.0
```

之后重新创建redis-master即可正确启动Pod了。

注意，除了pause镜像，其他Docker镜像也可能存在无法下载的情况，与上述情况类似，很可能也是网络配置使得镜像无法下载，解决方法同上。

2.Pod创建成功，但状态始终不是Ready，且RESTARTS的数量持续增加

在创建了一个RC之后，通过`kubectl get pods`命令查看Pod，发现如下情况：

```
.....
$ kubectl get pods
NAME          READY    STATUS    RESTARTS   AGE
zk-bg-ri3ru   0/1      Running   3           37s
.....
$ kubectl get pods
NAME          READY    STATUS    RESTARTS   AGE
zk-bg-ri3ru   0/1      Running   5           1m
.....
$ kubectl get pods
NAME          READY    STATUS    RESTARTS   AGE
zk-bg-ri3ru   0/1      ExitCode:0  6           1m
.....
$ kubectl get pods
NAME          READY    STATUS    RESTARTS   AGE
zk-bg-ri3ru   0/1      Running   7           1m
```

可以看到Pod已经创建成功了，但Pod的状态一会儿是Running，一会儿是ExitCode: 0，READY列中始终无法变成1，而且RESTARTS（重启的数量）的数量不断增加。

通常造成这种现象是因为容器的启动命令不能保持前台运行。

本例中的Docker镜像的启动命令为：

```
zkServer.sh start-background
```

在Kubernetes根据RC定义创建Pod后启动容器，容器的启动命令执行完成时，即认为该容器的运行已经结束，并且是成功结束（ExitCode=0）。然后，根据Pod的默认重启策略定义（RestartPolicy=Always），RC将启动这个容器。

新的容器执行启动命令后仍然会成功结束，然后RC会再次重启该容器，进入一个无限循环的过程中。

解决方法为将Docker镜像的启动命令设置为一个前台运行的命令，例如：

```
zkServer.sh start-foreground
```

5.3.5 寻求帮助

如果通过系统日志和容器日志都无法找到出现问题的原因，则还可以追踪源码进行分析，或者通过一些在线途径寻求帮助。

- Kubernetes 的常见问题参见 <https://github.com/GoogleCloudPlatform/kubernetes/wiki/User-FAQ>。
- Debugging 的常见问题参见 <https://github.com/GoogleCloudPlatform/kubernetes/wiki/Debugging-FAQ>。
- Service 的常见问题参见 <https://github.com/GoogleCloudPlatform/kubernetes/wiki/Services-FAQ>。
- StackOverflow 网站关于 Kubernetes 的主题参见 <http://stackoverflow.com/questions/tagged/kubernetes> 或 <http://stackoverflow.com/questions/tagged/google-container-engine>。
- IRC 频道（#google-containers）参见 <https://botbot.me/freenode/google-containers/>。
- Kubernetes 邮件列表 Email 参见 google-containers@googlegroups.com。

5.4 Kubernetes v1.3开发中的新功能

本节对Kubernetes v1.3版本的正在开发中的一些新功能进行介绍，包括Pet Set（用于管理有状态的容器应用）、init container（Pod中的初始化容器）、Cluster Federation（集群联邦管理）等内容。

5.4.1 PetSet（有状态的容器）

在Kubernetes集群中，组成一个微服务的后端Pod一般来说都是无状态的容器应用。例如通过RC来进行管理，只需要维持Pod的副本数量即可，当某个Pod失败时就直接销毁并重新创建一个新的Pod，提供能够水平扩展的微服务。我们可以称这类应用为“Cattle”（农场动物），即单个实例不是特别重要，可随时被替换。

但对于有状态的应用来说，即以集群的方式部署的大型应用软件，每个实例都需要具备唯一的标识，并且各个实例可能还有启动顺序的要求。v1.3版本新增了一种名为PetSet的资源对象，用于支持有状态的容器应用。我们可以称这类应用为“Pet”（宠物），即每个实例都非常重要，其身份不应被别的实例替换。

Pet Set能够确保为每个Pet设置一个唯一的身份标识，包括如下几种。

- 唯一且不变的hostname，并保存在DNS中。
- 唯一的顺序编号，用于确保各实例的启动顺序。
- 为每个容器提供永久存储，与其hostname和启动顺序绑定。

Pet Set能够用于许多应用场景，如下所述。

- 数据库应用，例如MySQL或PostgreSQL，其每个实例都需要挂载一个外部的永久存储。

- 集群化的应用软件，例如ZooKeeper、etcd、ElasticSearch等需要集群中的各成员有稳定的身份。

下面的例子描述了PetSet的创建和用法。

```
petset.yaml
# 使用headless Service，以创建相应Pod的DNS记录
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
    - port: 80
      name: web
    # *.nginx.default.svc.cluster.local
  clusterIP: None
  selector:
    app: nginx
---
# PetSet的定义
apiVersion: apps/v1alpha1
kind: PetSet
metadata:
  name: web
```

```
spec:
  serviceName: "nginx"
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
      annotations:
        pod.alpha.kubernetes.io/initialized: "true"
    spec:
      terminationGracePeriodSeconds: 0
      containers:
        - name: nginx
          image: gcr.io/google_containers/nginx-slim:0.8
          ports:
            - containerPort: 80
              name: web
          volumeMounts:
            - name: www
              mountPath: /usr/share/nginx/html
      volumeClaimTemplates:
        - metadata:
            name: www
            annotations:
              volume.alpha.kubernetes.io/storage-class:
```

anything

```
spec:
```

```
accessModes: [ "ReadWriteOnce" ]  
resources:  
  requests:  
    storage: 1Gi
```

在PetSet定义中，需要设置永久存储“volumeClaimTemplates”，需要系统管理员预先创建好外部PV（Persistent Volume），才能给PetSet使用。

使用kubectl create命令创建该PetSet:

```
$ kubectl create -f petset.yaml  
service "nginx" created  
petset "nginx" created
```

查看创建好的Pod的信息:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
web-0	1/1	Running	0	10m
web-1	1/1	Running	0	10m

可以看到每个Pod的名称不再是通过RC创建的带有一个UUID的名称，而是由“petset name”与一个序列号组成的特定名称：web-0和web-1。

而Pod名称也就是该Pod的hostname，即PetSet为每个Pet设置了稳定的主机名。

```
# kubectl exec web-0 -- sh -c 'hostname'
web-0

# kubectl exec web-1 -- sh -c 'hostname'
web-1
```

同时，每个Pod的网络身份也通过Service的定义被创建出来。根据Service的定义，该Service将在DNS中生成一条没有ClusterIP的记录：

```
nginx.default.svc.cluster.local
```

该Service的后端为两个Pet的地址（可以看成是Pet的服务名）：

```
web-0.nginx.default.svc.cluster.local
web-1.nginx.default.svc.cluster.local
```

这两个Pet的地址web-0.nginx和web-1.nginx将作为每个Pet的稳定网络身份被系统保存在DNS中，供客户端应用访问。

接下来通过一个busybox容器执行nslookup，验证Pet的服务地址：

```
# kubectl run -i --tty --image busybox dns-test --
restart=Never /bin/sh

Hit enter for command prompt

查询web-0.nginx:
/ # nslookup web-0.nginx
Server:      169.169.0.100
```

```
Address 1: 169.169.0.100
```

```
Name:      web-0.nginx
```

```
Address 1: 172.17.1.2
```

```
查询web-1.nginx:
```

```
/ # nslookup web-1.nginx
```

```
Server:     169.169.0.100
```

```
Address 1: 169.169.0.100
```

```
Name:      web-1.nginx
```

```
Address 1: 172.17.1.3
```

如果直接查询Nginx服务（headless service），则系统将返回后端两个Pet的IP地址列表：

```
/ # nslookup nginx
```

```
Server:     169.169.0.100
```

```
Address 1: 169.169.0.100
```

```
Name:      nginx
```

```
Address 1: 172.17.1.2
```

```
Address 2: 172.17.1.3
```

借助于headless service的功能，可以实现各Pet相互之间进行发现和访问。

当前版本**Pet Set**的使用限制如下。

- 只有**replicas**字段可以被更新，但更新后**Pet Set**仍然是按顺序依次创建各**Pet**。
- 删除**petset**时，系统不会自动删除已经运行中的**Pets**，需要手工删除。
- 出于数据安全的考虑，在删除**Pet**时系统不会自动删除该**Pet**使用的**PV**存储。
- 目前不支持**Pet**镜像的滚动升级操作，需要手工完成。

5.4.2 Init Container（初始化容器）

在很多应用场景中，应用在启动之前都需要一些初始化的操作，例如：

- 等待其他关联组件正确运行（例如数据库或某个后台服务）；
- 基于环境变量或配置模板生成配置文件；
- 从远程数据库获取本地所需配置，或者将自身注册到某个中央数据库；
- 下载相关依赖包，或者对系统进行一些预配置操作。

Kubernetes v1.3 版本引入了一个 Alpha 版本的新特性：`init container`，用于在启动普通容器之前启动一个或多个“初始化”容器，完成普通容器所需要的预置条件，如图5.22所示。`init container`与普通容器本质上是一样的，但它们是仅运行一次就结束的任务，并且必须成功执行完成后，系统才能继续执行下一个容器。根据Pod的重启策略（`RestartPolicy`），当 `init container` 执行失败，在设置了 `RestartPolicy=Never` 时，Pod 将会启动失败；而设置 `RestartPolicy=Always` 时，Pod 将会被系统自动重启。

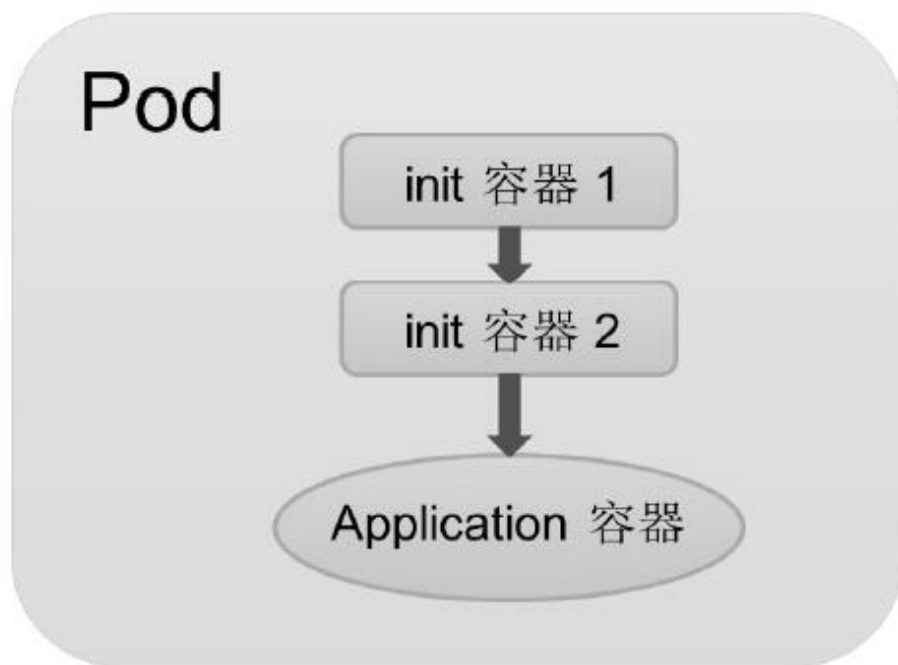


图5.22 init container

在当前的版本中要启用 init container 的配置，需要在 Pod 的 annotation 字段中设置 `pod.alpha.kubernetes.io/init-containers` 来定义需要执行的初始化容器列表。

下面，以 Nginx 应用为例，在启动 Nginx 之前，通过初始化容器 busybox 为 Nginx 创建一个 index.html 主页文件。

```
nginx-init-containers.yaml
```

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  name: nginx
```

```
  annotations:
```



```

    pod.alpha.kubernetes.io/init-containers: '[
      {
        "name": "install",
        "image": "busybox",
        "command": ["wget", "-O", "/work-
dir/index.html", "http://kubernetes.io/index.html"],
        "volumeMounts": [
          {
            "name": "workdir",
            "mountPath": "/work-dir"
          }
        ]
      }
    ]'

```

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

volumeMounts:

- name: workdir

mountPath: /usr/share/nginx/html

dnsPolicy: Default

volumes:

- name: workdir

emptyDir: {}

创建这个Pod:

```
# kubectl create -f nginx-init-containers.yaml  
pod "nginx" created
```

在运行init container的过程中，查看Pod的状态，可见Init过程还未完成:

```
# kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx	0/1	Init:0/1	0	1m

当init container成功执行完成，系统继续启动Nginx容器，再次查看Pod的状态:

```
# kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx	1/1	Running	0	7s

查看Pod的事件，可以看到系统按顺序运行Pod中的各个容器:

```
# kubectl describe pod nginx
```

Name: nginx

Namespace: default

..... (略)

Events:

FirstSeen	LastSeen	Count	From
-----------	----------	-------	------

SubobjectPath	Type		Reason
Message			
-----	-----	-----	----
-----	-----	-----	-----

4s	4s	1	{default-scheduler }
Scheduled	Successfully assigned nginx to k8s-node-1		Normal
4s	4s	1	{kubelet k8s-node-1} spec.initContainers{install}
Pulled	Container image "busybox" already present on machine		Normal
4s	4s	1	{kubelet k8s-node-1} spec.initContainers{install}
Created	Created container with docker id 81d3ef7ade94		Normal
4s	4s	1	{kubelet k8s-node-1} spec.initContainers{install}
Started	Started container with docker id 81d3ef7ade94		Normal
3s	3s	1	{kubelet k8s-node-1} spec.containers{nginx}
Pulled	Container image "nginx" already present on machine		Normal
3s	3s	1	{kubelet k8s-node-1} spec.containers{nginx}
Created	Created container with docker id 5a0bc53661f6		Normal
2s	2s	1	{kubelet k8s-

node-1}	spec.containers{nginx}	Normal
Started	Started container with docker id 5a0bc53661f6	

在init container的后续演进中，将进一步考虑Pod的资源使用限制、健康检查、镜像更新等问题。

5.4.3 ClusterFederation（集群联邦）

集群联邦是管理多个Kubernetes集群的集群，用于多个集群中的服务统一管理，以及当一个集群发生故障时，能够将业务恢复到其他集群上，如图5.23所示。目前集群联邦只能在谷歌或亚马逊的公有云上进行配置和使用。

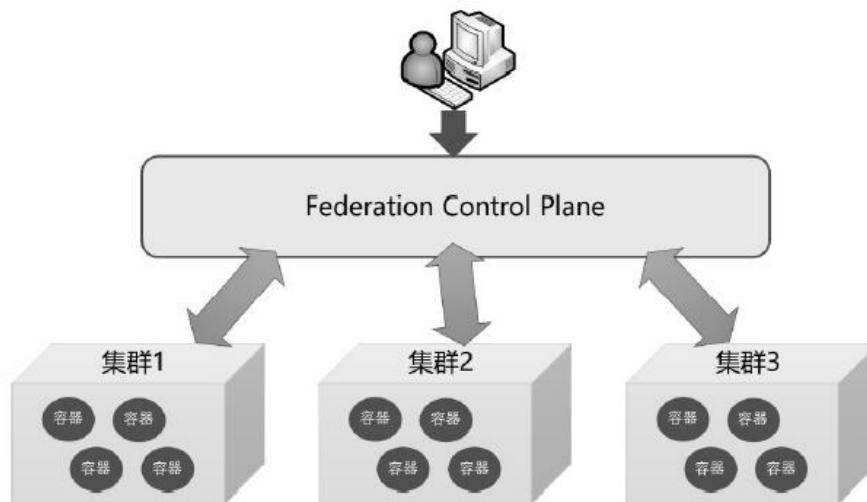


图5.23 集群联邦

集群联邦的主要组件由Federation Control Plane（控制平面）来完成对多个集群的管控，其核心组件包括federation-apiserver和federation-controller-manager和etcd，可以在已经存在的一个Kubernetes集群上以Pod的形式启动这些Federation组件。

下面以在GCE上运行一个ClusterFederation为例，创建federation-apiserver和federation-controller-manager的命令为：

```
$ KUBERNETES_PROVIDER=gce  
FEDERATION_DNS_PROVIDER=google-clouddns  
FEDERATION_NAME=myfederation  
DNS_ZONE_NAME=myfederation.example  
FEDERATION_PUSH_REPO_BASE=gcr.io/google_containers  
./federation/cluster/federation-up.sh
```

各个参数的含义如下。

- **KUBERNETES_PROVIDER**: 云服务商。
- **FEDERATION_DNS_PROVIDER**: 可以是 google-clouddns 或者 aws-route53。如果已经把KUBERNETES_PROVIDER设置为gce、gke及aws中的一个，那么系统会自动设置这一变量。该设置项用于为联邦服务提供域名解析能力。当联邦中的Kubernetes集群上的Pod或者Service发生变更的时候，会在DNS记录上做出相应的变更。
- **FEDERATION_NAME**: 联邦的名称，这一名称也会反映在DNS记录之中。
- **DNS_ZONE_NAME**: DNS记录的域名。用户需要购买和使用这个域名，让FEDERATION_DNS_PROVIDER能够为这一域名的查询提供正确的解析结果。

通过上面的命令会创建一个federation命名空间，并创建两个Deployment对象：federation-apiserver及federation-controller-manager。

验证创建出来的deployment:

\$ kubectl get deployments --namespace=federation				
NAME		DESIRED		CURRENT
UP-TO-DATE	AVAILABLE	AGE		
	federation-apiserver		1	1
1	1	1m		
	federation-controller-manager		1	1
1	1	1m		

federation-up.sh还会在kubeconfig中创建一个新纪录，用来和联邦APIServer进行通信。可以使用kubectlconfigview来查看。

另外，federation-up.sh创建的federation-apiserver Pod中包含的etcd容器使用了PV持久卷，用来提供持久化数据存储，目前只能在AWS、GKE 或 GCE 环境中进行创建。具体的PV设置可以通过修改federation/manifests/federation-apiserver-deployment.yaml来完成。

在联邦控制平面启动之后，就可以对现有的Kubernetes集群进行纳管了。

首先，我们需要创建一个secret对象，其中包含了Kubernetes集群的kubeconfig，联邦将会用这一内容和受管集群进行通信。假设kubeconfig文件位于/cluster1/kubeconfig，用下面的命令来创建secret:

```
$ kubectl create secret generic cluster1 --
namespace=federation --from-file=/cluster1/kubeconfig
```

文件名kubeconfig将用于设置secrete的key名称。

创建好secret之后，就可以注册集群了，一个集群对象的yaml配置文件如下：

```
apiVersion: federation/v1beta1
kind: Cluster
metadata:
  name: cluster1
spec:
  serverAddressByClientCIDRs:
    - clientCIDR: <client-cidr>
      serverAddress: <apiserver-address>
  secretRef:
    name: <secret-name>
```

需要把<client-cidr>、<apiserver-address>及<secret-name>替换为实际内容。<secret-name>是前面刚刚创建的secret的名称。serverAddressByClientCIDRs包含一系列地址，符合CIDR的客户端才能连接服务器的这一地址。我们可以设置服务器地址的CIDR为“0.0.0.0/0”，这样所有的客户端都可以访问。另外，如果希望内部客户端使用服务器的clusterIP，则可以把这一IP设置为serverAddress，然后设置clientCIDR为集群内的Pod地址范围。

将该yaml文件保存为/cluster1/cluster.yaml，运行下面的命令来进行集群的纳管：

```
$ kubectl create -f /cluster1/cluster.yaml --
context=federation-cluster
```

设置 `--context=federation-cluster` 意思是将请求发往联邦的 `federation-apiserver`。

查看纳管的结果：

\$ kubectl get clusters --context=federation-cluster			
NAME	STATUS	VERSION	AGE
cluster1	Ready		3m

当集群纳管之后，就可以使用集群联邦的功能了。

为了支持跨集群的服务发现机制，需要扩展KubeDNS服务，通过 `-federations` 参数设置集群联邦的总DNS服务：

<code>--federations=\${FEDERATION_NAME}=\${DNS_DOMAIN_NAME}</code>
--

可以通过编辑现有KubeDNS的RC中所包含的Pod模板来为Pod加入这一参数，删除当前运行的Pod之后，RC就会根据新的模板创建带有联邦DNS信息的KubeDNS服务了。

<code>\$ kubectl get rc --namespace=kube-system</code>
--

kube-dns的RC名是 `kube-dns-[id]` 的形式，用 `edit` 进行编辑：

<code>\$ kubectl edit rc <rc-name> --namespace=kube-system</code>

在弹出的yaml文件中加入 `--federation` 参数，保存退出，然后查询并删除现有的Pod。

```
$ kubectl delete pods <pod-name> --namespace=kube-system
```

至此，集群联邦设置完成，接下来就可以在多个集群上部署应用了。

假设当前已有4个集群纳管进了集群联邦：

```
$ kubectl get clusters --context=federation-cluster
```

NAME	STATUS	VERSION	AGE
cluster1	Ready		3m
cluster2	Ready		3m
cluster3	Ready		3m
cluster4	Ready		3m

在这4个集群上创建Nginx服务：

```
$ kubectl --context=federation-cluster create -f services/nginx.yaml
```

等待该服务在所有集群上创建完成，并且集群联邦内的服务也更新完成，通常这需要几分钟。

查看服务的状态，可以看到在每个集群上创建的Loadbalancer的IP地址：

```
$ kubectl --context=federation-cluster describe services nginx
```

```

Name:          nginx
Namespace:     default
Labels:        run=nginx
Selector:      run=nginx
Type:          LoadBalancer
IP:
               LoadBalancer Ingress:      104.197.246.190,
130.211.57.243, 104.196.14.231, 104.199.136.89
Port:          http      80/TCP
Endpoints:     <none>
Session Affinity:  None
No events.

```

登录其中一个集群，查看由federation-controller-manager创建的Nginx服务：

```

$ kubectl --context=cluster1 get svc nginx

```

	NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	
AGE					
	nginx	10.63.250.98	104.199.136.89	80/TCP	9m

可以看到集群联邦通过在每个集群上创建一个同名的服务，但此时由于每个集群的Service后端还没有运行任何Pod，所以这些联邦服务还无法正常工作。

接下来通过在每个集群上运行Pod来支撑Service：

```
$ kubectl --context=cluster1 run nginx --
image=nginx:1.11.1-alpine --port=80

$ kubectl --context=cluster2 run nginx --
image=nginx:1.11.1-alpine --port=80

$ kubectl --context=cluster3 run nginx --
image=nginx:1.11.1-alpine --port=80

$ kubectl --context=cluster4 run nginx --
image=nginx:1.11.1-alpine --port=80
```

当这些Pod成功运行后，Service将被集群联邦设置为正常状态，然后集群联邦将会配置相应的公共DNS记录。假设使用的是Google Cloud DNS，并且DNS域名为example.com，则可以看到每个集群上的Nginx服务都被设置好了一个DNS名，可供其他应用访问时使用：

```
$ gcloud dns managed-zones describe example-dot-com
creationTime: '2016-06-26T18:18:39.229Z'
description: Example domain for Kubernetes Cluster
Federation
dnsName: example.com.
id: '3229332181334243121'
kind: dns#managedZone
name: example-dot-com
nameServers:
- ns-cloud-a1.googledomains.com.
- ns-cloud-a2.googledomains.com.
- ns-cloud-a3.googledomains.com.
- ns-cloud-a4.googledomains.com.
```

```
$ gcloud dns record-sets list --zone example-dot-com
```

			NAME
TYPE	TTL	DATA	
			example.com.
NS	21600	ns-cloud-e1.googledomains.com., ns-cloud-e2.googledomains.com.	
			example.com.
SOA	21600	ns-cloud-e1.googledomains.com. cloud-dns-hostmaster.google.com. 1 21600 3600 1209600 300	
			nginx.mynamespace.myfederation.svc.example.com.
A	180	104.XXX.XXX.XXX, 130.XXX.XX.XXX, 104.XXX.XX.XXX, 104.XXX.XXX.XX	
			nginx.mynamespace.myfederation.svc.cluster1.example.com.
A	180	104.XXX.XXX.XXX	
			nginx.mynamespace.myfederation.svc.cluster2.example.com.
A	180	104.XXX.XXX.XXX, 104.XXX.XXX.XXX, 104.XXX.XXX.XXX	
			nginx.mynamespace.myfederation.svc.cluster3.example.com.
A	180	130.XXX.XX.XXX	
			nginx.mynamespace.myfederation.svc.cluster4.example.com.
A	180	130.XXX.XX.XXX, 130.XXX.XX.XXX	

nginx.mynamespace.myfederation.svc.cluster4.example.com.

CNAME 180

nginx.mynamespace.myfederation.svc.example.com.

.....

第6章 Kubernetes源码导读

6.1 Kubernetes源码结构和编译步骤

Kubernetes 的源码现在托管在 GitHub 上，地址为 <https://github.com/googlecloudplatform/kubernetes>。

编译脚本存放在build子目录下，在Linux环境（可以是虚拟机）中执行如下命令即可完成代码的编译过程：

```
git clone
https://github.com/GoogleCloudPlatform/kubernetes.git
cd kubernetes/build
./release.sh
```

制作release的过程其实有不少有意思的事情发生，包括启动Docker容器来安装Go语言环境、etcd等，读者若有兴趣则可以查看release.sh脚本。另外，如果编译环境是通过HTTP代理上网的，则需要设置好Git与Docker相关的HTTP代理参数，同时在文件kubernetes/build/build-image/Dockerfile中增加如下HTTP代理参数。

- ENV `http_proxy=http://username : password@proxyaddr : proxyport` °
- ENV `https_proxy=http://username : password@proxyaddr : proxyport` °

在编译过程中产生的与Docker相关的docker image、dockerfile及编译好的二进制文件包，则存放在kubernetes/_output目录下，这个目录总共有4个子目录：dockerized、images、release-stage、release-tars，我们关心后两个目录，其中release-stage目录下存放的是支持linux-amd64架构的Server端的二进制可执行文件（放在server子目录下），以及支持不同平台的Client端的二进制可执行文件（放在client子目录下），release-tars则存放的是release-stage目录下各级子目录的压缩包，与从官方网站下载的完全一样。

考虑到学习和调试Kubernetes代码的便利性，我们接下来介绍如何在Windows的LiteIDE开发环境中完成Kubernetes代码的编译和调试。本文假设Windows上的GO运行时框架和LiteIDE开发环境已经建立好，并通过 `git clone` 命令已经将 `https://github.com/GoogleCloudPlatform/kubernetes.git` 下载到本地 C:\kubernetes 目录中。通过分析 Kubernetes 的目录结构，我们发现 Kubernetes 的源码都在 pkg 子目录下。接下来建立 k8s 工程目录，目录位置为 C:\project\go\k8s，并在里面建立 src、pkg 两个子目录，然后把 C:\kubernetes\Godeps_workspace\src 全部转移到 C:\project\go\k8s\src 目录下，因为这里是 Kubernetes 源码的所有依赖包，所以如果手动一个一个地下载，则恐怕以国内的网速一天也搞不定。转移完成后，C:\project\go\k8s\src 的目录结构包括如下内容：

```
C:\project\go\k8s\src>dir
2015-07-14  11:56    <DIR>          bitbucket.org
2015-07-14  11:56    <DIR>          code.google.com
2015-07-17  12:30    <DIR>          github.com
2015-07-14  11:56    <DIR>          golang.org
2015-07-14  11:56    <DIR>          google.golang.org
2015-07-14  11:56    <DIR>          gopkg.in
2015-07-14  11:56    <DIR>          speter.net
```

接下来把 C : \kubernetes 的整个目录移动到 C : \project\go\k8s\src\github.com\GoogleCloudPlatform\下，因为Kubernetes的源码包的完整名字为“github.com/GoogleCloudPlatform/kubernetes/pkg”。上述工作完成以后，所有的源码都在C: \project\go\k8s\src目录下了，我们用LiteIDE打开C: \project\go\k8s，单击菜单“查看”→“管理Gopath”→添加目录“C: \project\go\k8s”，然后可以进入目录github.com/GoogleCloudPlatform/kubernetes/pkg下，逐一编译每个package目录了，如图6.1所示。

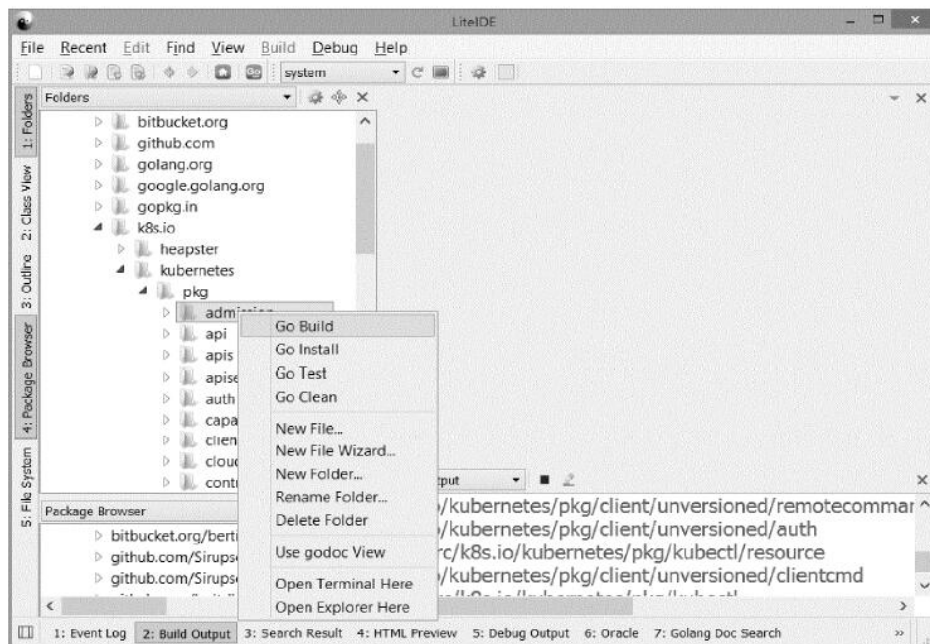


图6.1 LiteIDE编译Kubernetes的package

在每个package都编译完成以后，我们可以尝试启动kube-scheduler进程：在 LiteIDE 里 打开 github.com/GoogleCloudPlatform/kubernetes/pkg/plugin/cmd/kube-scheduler/scheduler.go，并且按快捷键Ctrl+R，你会惊奇地发现这个Kubernetes服务器端进程竟然也能在Windows下运行起来。以下是 LiteIDE输出的控制台日志：

```
c:/go/bin/go.exe    build    -i
[C:/project/go/k8s/src/github.com/GoogleCloudPlatform/kuber
netes/plugin/cmd/kube-scheduler]
```

成功：进程退出代码 0。

```
C:/project/go/k8s/src/github.com/GoogleCloudPlatform/kubern
```

```
etes/plugin/cmd/kube-scheduler/kube-scheduler.exe
[C:/project/go/k8s/src/github.com/GoogleCloudPlatform/kuber
netes/plugin/cmd/kube-scheduler]
W0717 16:05:26.742413 11344 server.go:83] Neither --
kubeconfig nor --master was specified. Using default API
client. This might not work.
E0717 16:05:27.747413 11344 reflector.go:136] Failed
to list *api.Node: Get http://localhost:8080/api/v1/nodes
fieldSelector=spec.unschedulable%3Dfalse: dial tcp
127.0.0.1:8080: ConnectEx tcp: No connection could be made
because the target machine actively refused it.
E0717 16:05:27.748413 11344 reflector.go:136] Failed
to list *api.Pod: Get http://localhost:8080/api/v1/pods
fieldSelector=spec.nodeName%21%3D: dial tcp 127.0.0.1:8080:
ConnectEx tcp: No connection could be made because the
target machine actively refused it.
```

在Kubernetes的源码里包括不少单元测试，你可以在LiteIDE里运行通过，但有部分测试代码目前在Windows上无法通过，毕竟Kubernetes是为Linux打造的。接下来我们分析下Kubernetes源码的整体结构，Kubernetes的源码总体分为pkg、cmd、plugin、test等顶级package，其中pkg为Kubernetes的主体代码，cmd为Kubernetes所有后台进程的代码（如kube-apiserver进程、kube-controller-manager进程、kube-proxy进程、kubelet进程等），plugin则包括一些插件及kube-scheduler的代码，test包是Kubernetes的一些测试代码。

从总体来看，Kubernetes 1.0的当前包结构还是有点乱，开源团队还在继续优化中，可以从源码的TODO注释中看出这一点。表6.1给出了Kubernetes当前主要package的源码分析结果。

表6.1 Kubernetes主要package的源码分析结果

package	模 块 用 途	类 数 量
admission	权限控制框架，采用了责任链模式、插件机制	少
api	Kubernetes 所提供的 Rest API 接口的相关类，例如接口数据结构相关的 MetaData 结构、Volume 结构、Pod 结构、Service 结构等，以及数据格式验证转换工具类等，由于 API 是分版本的，所以这里是每个版本一个子 Package，例如 v1beta、v1 及 latest	中
apiserver	实现了 HTTP Rest 服务的一个基础性框架，用于 Kubernetes 的各种 Rest API 的实现，在 apiserver 包里也实现了 HTTP Proxy，用于转发请求（到其他组件，比如 Minion 节点上）	中
auth	3A 认证模块，包括用户认证、鉴权的相关组件	少

续表

package	模块用途	类数量
client	是 Kubernetes 中公用的客户端部分的相关代码, 实现协议为 HTTP Rest, 用于提供一个具体的操作, 例如对 Pod、Service 等的增删改查, 这个模块也定义了 kubeletClient, 同时为了高效地进行对象查询, 此模块也实现了一个带缓存功能的存储接口 Store	多
cloudprovider	定义了云服务提供商运行 Kubernetes 所需的接口, 包括 TCPLoadBalancer 的获取和创建; 获取当前环境中的节点列表 (节点是一个云主机) 和节点的具体信息; 获取 Zone 信息; 获取和管理路由的接口等, 默认实现了 AWS、GCE、Mesos、OpenStack、RackSpace 等云服务供应商的接口	中
controller	这部分提供了资源控制器的简单框架, 用于处理资源的添加、变更、删除等事件的派发和执行, 同时实现了 Kubernetes 的 ReplicationController 的具体逻辑	少
kubectrl	Kubernetes 的命令行工具 kubectrl 的代码模块, 包括创建 Pod、服务、Pod 扩容、Pod 滚动升级等各种命令的具体实现代码	多
kubelet	Kubernetes 的 kubelet 的代码模块, 是 Kubernetes 的核心模块之一, 定义了 Pod 容器的接口, 提供了 Docker 与 Rkt 两种容器实现类, 完成了容器及 Pod 的创建, 以及容器状态的监控、销毁、垃圾回收等功能	多
master	Kubernetes 的 Master 节点代码模块, 创建 NodeRegistry、PodRegistry、ServiceRegistry、EndpointRegistry 等组件, 并且启动 Kubernetes 自身的相关服务, 服务的 ClusterIP 地址分配及服务的 NodePort 端口分配, 也是在这里完成的	少
proxy	Kubernetes 的服务代理和负载均衡相关功能的模块代码, 目前实现了 round-robin 的负载均衡算法	少
registry	Kubernetes 的 NodeRegistry、PodRegistry、ReplicationControllerRegistry、ServiceRegistry、EndpointRegistry、PersistentVolumeRegistry 等注册表服务的接口及对应 Rest 服务的相关代码	多
runtime	为了让多个 API 版本共存, 需要采用一些设计来完成不同 API 版本的数据结构的转换, API 中数据对象的 Encode/Decode 逻辑也最好集中化, Runtime 包就是为了这个目的而设计的	少
volume	实现了 Kubernetes 的各种 Volume 类型, 分别对应亚马逊 ESB 存储、谷歌 GCE 的存储、Linux Host 目录存储、GlusterFS 存储、iSCSI 存储、NFS 存储、RBD 存储等, volume 包同时实现了 Kubernetes 容器的 Volume 卷的挂载、卸载功能	多
cmd	包括了 Kubernetes 所有后台进程的代码 (如 kube-apiserver 进程、kube-controller-manager 进程、kube-proxy 进程、kubelet 进程等), 而这些进程具体的业务逻辑代码则都在 pkg 中实现了	
plugin	子包 cmd/kuber-scheduler 实现了 Schedule Server 的框架, 用于执行具体的 Scheduler 的调度, pkg/admission 子包则实现了 Admission 权限框架的一些默认实现类, 例如 alwaysAdmit、alwaysDeny 等; pkg/auth 子包实现了权限认证框架 (auth 包的) 里定义的认证接口类, 例如 HTTP BasicAuth、X509 证书认证; pkg/scheduler 子包则定义了一些具体的 Pod 调度器 (Scheduler)	中

6.2 kube-apiserver进程源码分析

Kubernetes APIServer是由kube-apiserver进程实现的，它运行在Kubernetes的管理节点——Master上并对外提供Kubernetes Restful API服务，它提供的主要是与集群管理相关的API服务，例如校验pod、service、replication controller的配置并存储到后端的etcd Server上。下面我们分别对其启动过程、关键代码分析及设计总结等进行深入讲解。

6.2.1 进程启动过程

kube-apiserver进程的入口类源码位置如下:

```
github.com/GoogleCloudPlatform/kubernetes/cmd/kube-apiserver/apiserver.go
```

入口main () 函数的逻辑如下:

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())
    rand.Seed(time.Now().UTC().UnixNano())

    s := app.NewAPIServer()
    s.AddFlags(pflag.CommandLine)

    util.InitFlags()
    util.InitLogs()
    defer util.FlushLogs()

    verflag.PrintAndExitIfRequested()

    if err := s.Run(pflag.CommandLine.Args()); err !=
nil {
        fmt.Fprintf(os.Stderr, "%v\n", err)
```

```
        os.Exit(1)
    }
}
```

上述代码的核心为下面三行，创建一个API Server结构体并将命令行启动参数传入，最后启动监听：

```
s := app.NewAPIServer()
s.AddFlags(pflag.CommandLine)
s.Run(pflag.CommandLine.Args())
```

我们先来看看都有哪些常用的命令行参数被传递给了API Server对象，下面是运行在Master节点的kube-apiserver进程的命令行信息：

```
/usr/bin/kube-apiserver --logtostderr=true --
etcd_servers=http://127.0.0.1:4001 --address=0.0.0.0 --
port=8080 --kubelet_port=10250 --allow_privileged=false --
service-cluster-ip-range=10.254.0.0/16
```

可以看到关键的几个参数有etcd_servers的地址、API Server绑定和监听的本地地址、kubelet的运行端口及Kubernetes服务的clusterIP地址。

下面是app.NewAPIServer()的代码，我们看到这里的控制还是很全面的，包括安全控制（CertDirectory、HTTPS默认启动）、权限控制（AuthorizationMode、AdmissionControl）、服务限流控制（APIRate、APIBurst）等，这些逻辑说明了API Server是按照企业级平台的标准所设计和实现的。


```
func NewAPIServer() *APIServer {  
    s := APIServer{  
        InsecurePort:      8080,  
        InsecureBindAddress:  
            util.IP(net.ParseIP("127.0.0.1")),  
        BindAddress:  
            util.IP(net.ParseIP("0.0.0.0")),  
        SecurePort:       6443,  
        APIRate:          10.0,  
        APIBurst:         200,  
        APIPrefix:        "/api",  
        EventTTL:         1 * time.Hour,  
        AuthorizationMode: "AlwaysAllow",  
        AdmissionControl:  "AlwaysAdmit",  
        EtcdPathPrefix:   master.DefaultEtcdPathPrefix,  
        EnableLogsSupport: true,  
        MasterServiceNamespace: api.NamespaceDefault,  
        ClusterName:        "kubernetes",  
        CertDirectory:      "/var/run/kubernetes",  
  
        RuntimeConfig: make(util.ConfigurationMap),  
        KubeletConfig: client.KubeletConfig{  
            Port: ports.KubeletPort,  
            EnableHttps: true,  
            HTTPTimeout: time.Duration(5) *
```

```
time.Second,  
        },  
    }  
  
    return &s  
}
```

创建了API Server结构体实例后，apiserver.go将此实例传入子包app/server.go的func (s*API Server) Run ([]string)方法里，最终绑定本地端口并创建一个HTTP Server与一个HTTPS Server，从而完成整个进程的启动过程。

Run方法的代码有很多，这里就不再列出源码，对该方法的源码解读如下。

(1) 调用verifyClusterIPFlags方法，验证ClusterIP参数是否已设置及是否有效。

(2) 验证etcd-servers的参数是否已设置。

(3) 如果初始化CloudProvider，且没有CloudProvider的参数，则日志告警并继续。

(4) 根据KubeletConfig的配置参数，调用pkg/Client/kubeclient.go中的方法NewKubeletClient () 创建一个kubelet Client对象，这其实是一个 HTTPKubeletClient 实例，目前只用于 kubelet 的健康检查 (KubeletHealthChecker) 。

(5) 判断哪些API Version需要关闭，目前在1.0代码中默认关闭了v1beta3的API版本。

(6) 创建一个Kubernetes的RestClient对象，具体的代码在pkg/client/helper.go的TransportFor（）方法里完成，通过它完成Pod、Replication Controller及Kubernetes Service等对象的CRUD操作。

(7) 创建用于访问etcd Server的客户端，具体代码在newEtcd（）方法里实现，从代码调用中可以看出，Kubernetes采用的是github.com/coreos/go-etcd/client.go这个客户端实现。

(8) 建立鉴权（Authenticator）、授权（Authorizer）、服务许可框架和插件（AdmissionControl）的相关代码逻辑。

(9) 获取和设置APIServer的ExternalHost的名称，如果没有提供ExternalHost参数，且Kubernetes运行在谷歌的GCE云平台上，则尝试通过CloudProvider接口获取本机节点的外部IP地址。

(10) 如果运行在云平台中，则安装本机的SSH Key到Kubernetes集群中的所有虚拟机上。

(11) 用APIServer的数据及上述过程中创建的一些对象（KubeletClient、etcdClient、authenticator、admissionController等）作为参数，构造Kubernetes Master的Config结构（pkg/master/master.go），以此生成一个Master实例，具体代码在master.go中的New（c*Config）方法里。

(12) 用上述创建的Master实例，分别创建HTTP Server及安全的HTTPS Server来开始监听客户端的请求，至此整个进程启动完毕。

6.2.2 关键代码分析

在6.2.1节里对kube-apiserver进程的启动过程进行了详细分析，我们发现Kubernetes API Service的关键代码就隐藏在pkg/master/master.go里，APIServer这个结构体只不过是一个参数传递通道而已，它的数据最终传给了pkg/master/master.go里的Master结构体，下面是它的完整定义：

```
// Master contains state for a Kubernetes cluster
master/api server.

type Master struct {
    // "Inputs", Copied from Config
    serviceClusterIPRange *net.IPNet
    serviceNodePortRange  util.PortRange
    cacheTimeout           time.Duration
    minRequestTimeout      time.Duration

    mux                apiserver.Mux
    muxHelper          *apiserver.MuxHelper
    handlerContainer    *restful.Container
    rootWebService      *restful.WebService
    enableCoreControllers bool
    enableLogsSupport    bool
    enableUISupport      bool
}
```

```
enableSwaggerSupport    bool
enableProfiling         bool
apiPrefix               string
corsAllowedOriginList   util.StringList
authenticator           authenticator.Request
authorizer              authorizer.Authorizer
admissionControl        admission.Interface
masterCount             int
v1beta3                 bool
v1                      bool
requestContextMapper    api.RequestContextMapper
```

```
    // External host is the name that should be used
in external (public internet) URLs for this master
```

```
externalHost string
```

```
    // clusterIP is the IP address of the master
within the cluster.
```

```
clusterIP          net.IP
```

```
publicReadWritePort int
```

```
serviceReadWriteIP net.IP
```

```
serviceReadWritePort int
```

```
masterServices      *util.Runner
```

```
    // storage contains the RESTful endpoints exposed
by this master
```

```
storage map[string]rest.Storage
```

```
    // registries are internal client APIs for
```

accessing the storage layer

// TODO: define the internal typed interface in a way that clients can

// also be replaced

nodeRegistry minion.Registry

namespaceRegistry namespace.Registry

serviceRegistry service.Registry

endpointRegistry endpoint.Registry

serviceClusterIPAllocator service.RangeRegistry

serviceNodePortAllocator service.RangeRegistry

// "Outputs"

Handler http.Handler

InsecureHandler http.Handler

// Used for secure proxy

dialer apiserver.ProxyDialerFunc

tunnels *util.SSHTunnellList

tunnelsLock sync.Mutex

installSSHKey InstallSSHKey

lastSync int64 // Seconds since Epoch

lastSyncMetric prometheus.GaugeFunc

clock util.Clock

}

在这段代码里，除了之前我们熟悉的那些变量，又多了几个陌生的重要变量，接下来我们逐一对其进行分析讲解。

首先是类型为`apiserver.Mux`（来自文件`pkg/apiserver/apiserver.go`）的`mux`变量，下面是对它的定义：

```
// mux is an object that can register http handlers.
type Mux interface {
    Handle(pattern string, handler http.Handler)
    HandleFunc(pattern string, handler
func(http.ResponseWriter, *http.Request))
}
```

如果你熟悉Socket编程，特别使用过或者研究过HTTP Rest的一些框架，那么对于这个Mux接口就再熟悉不过了，它是一个HTTP的多分器（Multiplexer），其实它也是Golang HTTP基础包里的`http.ServeMux`的一个接口子集，用于派发（Dispatch）某个Request路径（这里用`pattern`变量表示）到对应的`http.Handler`进行处理。实际上在`master.go`代码中是生成一个`http.ServeMux`对象并赋值给`apiserver.Mux`变量，在代码中还有强制类型转换的语句。从上述分析来看，`apiserver.Mux`的引入是设计的一个败笔，并没有增加什么价值，反而增加了理解代码的难度。此外，为了更好地实现Rest服务，Kubernetes在这里引入了一个第三方的REST框架：github.com/emicklei/go-restful。

`go-restful`在GitHub上有36个贡献者，采用了“路由”映射的设计思想，并且在API设计中使用了流行的Fluent Style风格，使用起来酣畅淋漓，也难怪Kubernetes选择了它。下面是`go-restful`的优良特性。

- Ruby on Rails 风格的 Rest 路由映射，例如`/people/{person_id}/groups/{group_id}`。
- 大大简化了Rest API的开发工作。

- 底层实现采用Golang的HTTP协议栈，几乎没有限制。
- 拥有完整的单元包代码，很容易开发一个可测试的Rest API。
- Google AppEngine ready。

go-restful框架中的核心对象如下。

- `restful.Container`：代表一个 HTTP Rest 服务器，包括一组 `restful.WebService` 对象和一个 `http.ServeMux` 对象，使用 `RouteSelector` 进行请求派发。
- `restful.WebService`：表示一个 Rest 服务，由多个 Rest 路由（`restful.Route`）组成，这一组Rest路由共享同一个Root Path。
- `restful.Route`：表示一个Rest路由，Rest路由主要由Rest Path、HTTP Method、输入输出类型（HTML/JSON）及对应的回调函数 `restful.RouteFunction` 组成。
- `restful.RouteFunction`：一个用于处理具体的REST调用的函数接口定义，具体定义为 `type RouteFunction func (*Request, *Response)`。

Master结构体里包含了对`restful.Container`与`restful.WebService`这两个go-restful核心对象的引用，在接下来的Master对象的构造方法中（对应代码为`master.go`的`func New (c*Config) *Master`）被初始化。那么，问题又来了，Kubernetes的这么一堆Rest API又是在哪里定义的，是如何被绑定到`restful.Route`里的呢？

要理解这个问题，我们要首先弄清楚Master结构体中的变量：

```
storage map[string]rest.Storage
```

storage 变量 是一个 Map， Key 为 Rest API 的 path， Value 为 rest.Storage接口， 此接口是一个通用的符合Restful要求的资源存储服务接口， 每个服务接口负责处理一类（Kind） Kubernetes Rest API中的数据对象——资源数据， 只有一个接口方法： New（）， New（）方法返回该Storage服务所能识别和管理的某种具体的资源数据的一个空实例。

```
type Storage interface {  
    New() runtime.Object  
}
```

在运行期间， Kubernetes API Runtime运行时框架会把New（）方法返回的空对象的指针传入 Codec.DecodeInto（[]byte， runtime.Object）方法中， 从而完成HTTP Rest请求中的Byte数组反序列化逻辑。 Kubernetes API Server中所有对外提供服务的Restful资源都实现了此接口， 这些资源包括 pods、 bindings、 podTemplates、 replicationControllers、 services等， 完整的列表就在master.go的func（m*Master） init（c*Config）中， 下面是相关代码片段（截取部分代码）。

```
m.storage = map[string]rest.Storage{  
    "pods": podStorage.Pod,  
    "pods/status": podStorage.Status,  
    "pods/log": podStorage.Log,  
    "pods/exec": podStorage.Exec,  
    "pods/portforward": podStorage.PortForward,  
    "pods/proxy": podStorage.Proxy,
```

```

        "pods/binding":      podStorage.Binding,
        "bindings":         podStorage.Binding,

        "podTemplates": podTemplateStorage,

        "replicationControllers": controllerStorage,
                                "services":
service.NewStorage(m.serviceRegistry,      m.nodeRegistry,
m.endpointRegistry,      serviceClusterIPAllocator,
serviceNodePortAllocator, c.ClusterName),
        "endpoints":      endpointsStorage,
        "minions":      nodeStorage,

```

看到上面这段代码，你在潜意识里已经明白，这其实就是似曾相识的Kubernetes Rest API列表，storage这个Map的Key就是Rest API的访问路径，Value却不是之前说好的restful.Route。聪明的你一定想到了答案：必然存在一个“转换适配”的方法来实现上述转换！这段不难理解但源码超长的方法就在pkg/apiserver/api_installer.go的下述方法里：

```

func (a *APIInstaller) registerResourceHandlers(path
string,  storage  rest.Storage,  ws  *restful.WebService,
proxyHandler http.Handler)

```

上述方法把一个 path 对应的 rest.Storage 转换成一系列的 restful.Route 并添加到指针 restful.WebService 中。这个函数的代码之所以很长，是因为有各种情况要考虑，比如 pods/portforward 这种路径要处理 child，还要判断每种的 Storage 资源类型所支持的操作类型；比如

是否支持create、delete、update及是否支持list、watch、patcher操作等，对各种情况都考虑以后，这个函数的代码量已接近500行！估计Kubernetes这段代码的作者也不大好意思，于是外面封装了简单函数：`func (a*APIInstaller) Install`，内部循环调用`registerResourceHandlers`，返回最终的`restful.WebService`对象，此方法的主要代码如下：

```
// Installs handlers for API resources.
func (a *APIInstaller) Install() (ws
*restful.WebService, errors []error) {
    // Register the paths in a deterministic (sorted)
order to get a deterministic swagger spec.
    paths := make([]string, len(a.group.Storage))
    var i int = 0
    for path := range a.group.Storage {
        paths[i] = path
        i++
    }
    sort.Strings(paths)
    for _, path := range paths {
        if err := a.registerResourceHandlers(path,
a.group.Storage[path], ws, proxyHandler); err != nil {
            errors = append(errors, err)
        }
    }
    return ws, errors
}
```

为了区分API的版本，在apiserver.go里定义了一个结构体：APIGroupVersion。以下是其代码：

```
type APIGroupVersion struct {
    Storage map[string]rest.Storage
    Root     string
    Version  string

    // ServerVersion controls the Kubernetes
    APIVersion used for common objects in the apiserver
    // schema like api.Status, api.DeleteOptions, and
    api.ListOptions. Other implementors may
    // define a version "v1beta1" but want to use the
    Kubernetes "v1beta3" internal objects. If
    // empty, defaults to Version.
    ServerVersion string

    Mapper meta.RESTMapper

    Codec      runtime.Codec
    Typer      runtime.ObjectTyper
    Creator    runtime.ObjectCreator
    Convertor  runtime.ObjectConvertor
    Linker     runtime.SelfLinker

    Admit      admission.Interface
    Context    api.RequestContextMapper
}
```

```

        ProxyDialerFn    ProxyDialerFunc
        MinRequestTimeout time.Duration
    }

```

我们注意到APIGroupVersion是与rest.Storage Map捆绑的，并且绑定了相应版本的Codec、Convertor用于版本转换，这样就很容易理解Kubernetes是怎样区分多版本API的Rest服务的。以下是过程详解。

首先，在APIGroupVersion的InstallREST（container*restful.Container）方法里，用Version变量来构造一个Rest API Path前缀并赋值给APIInstaller的prefix变量，并调用它的Install（）方法完成Rest API的转换，代码如下：

```

func (g *APIGroupVersion) InstallREST(container
*restful.Container) error {
    info :=
&APIRequestInfoResolver{util.NewStringSet(strings.TrimPrefix(g.Root, "/")), g.Mapper}
    prefix := path.Join(g.Root, g.Version)
    installer := &APIInstaller{
        group:      g,
        info:        info,
        prefix:      prefix,
        minRequestTimeout: g.MinRequestTimeout,
        proxyDialerFn: g.ProxyDialerFn,
    }

```

```
ws, registrationErrors := installer.Install()

container.Add(ws)
```

接着，在APIInstaller的Install（）方法里用prefix（API版本）前缀生成WebService的相对根路径：

```
func (a *APIInstaller) newWebService()
*restful.WebService {
    ws := new(restful.WebService)
    ws.Path(a.prefix)
    ws.Doc("API at"+ a.prefix +"version"+ a.group.Version)
    // TODO: change to restful.MIME_JSON when we set
content type in client
    ws.Consumes("*/")
    ws.Produces(restful.MIME_JSON)
    ws.ApiVersion(a.group.Version)

    return ws
}
```

最后，在Kubernetes的Master初始化方法func（m*Master）init（c*Config）里生成不同的APIGroupVersion对象，并调用InstallRest（）方法，完成最终的多版本API的Rest服务装配流程：

```
if m.v1beta3 {

                                if err :=
m.api_v1beta3().InstallREST(m.handlerContainer); err != nil
```

```

{
    glog.Fatalf("Unable to setup API v1beta3:
    %v", err)
}
    apiVersions = append(apiVersions, "v1beta3")
}
    if m.v1 {
        if err :=
m.api_v1().InstallREST(m.handlerContainer); err != nil {
            glog.Fatalf("Unable to setup API v1: %v",
err)
        }
        apiVersions = append(apiVersions, "v1")
    }
}

```

至此，Rest API的多版本问题还有最后一个需要澄清，即在不同的版本中接口的输入输出参数的格式是有差别的，Kubernetes是怎么处理这个问题的？

要弄明白这一点，我们首先要研究KubernetesAPI里的数据对象的序列化、反序列化的实现机制。为了同时解决数据对象的序列化、反序列化与多版本数据对象的兼容和转换问题，Kubernetes设计了一套复杂的机制，首先，它设计了 `conversion.Scheme` 这个结构体（`pkg/conversion/schema.go`里），以下是对它的定义：

```

// Scheme defines an entire encoding and decoding
scheme.

```

```

type Scheme struct {
    // versionMap allows one to figure out the go type
of an object          //with the given version and name.
    versionMap map[string]map[string]reflect.Type
    // typeToVersion allows one to figure out the
version for a given //go object  The reflect.Type we index
by should *not* be a pointer. If the same type
    // is registered for multiple versions, the last
one wins.

    typeToVersion map[reflect.Type]string
    // typeToKind allows one to figure out the desired
"kind" field //for a given go object. Requirements and
caveats are the same as typeToVersion.
    typeToKind map[reflect.Type][]string
    // converter stores all registered conversion
functions. It also //has default coverting behavior.
    converter *Converter
    // cloner stores all registered copy functions. It
also has default
    // deep copy behavior.
    cloner *Cloner
    // Indent will cause the JSON output from Encode
to be indented, iff it is true.
    Indent bool
    // InternalVersion is the default internal
version. It is recommended that
    // you use "" for the internal version.

```



```

        InternalVersion string

        // MetaInsertionFactory is used to create an
        object to store and retrieve

        // the version and kind information for all
        objects. The default // uses the keys "apiVersion" and
        "kind" respectively.

        MetaFactory MetaFactory
    }

```

在上述代码中可以看到，`typeToVersion`与`versionMap`属性是为了解决数据对象的序列化与反序列化问题，`converter`属性则负责不同版本的数据对象转换问题，**Kubernetes**这个设计思路简单方便地解决了多版本的序列化和数据转换问题，不得不赞！下面是`conversion.Scheme`里序列化、反序列化的核心方法`NewObject()`的代码：通过查找`versionMap`里匹配的注册类型，以反射方式生成一个空的数据对象：

```

func (s *Scheme) NewObject(versionName, kind string)
(interface{}, error) {
    if types, ok := s.versionMap[versionName]; ok {
        if t, ok := types[kind]; ok {
            return reflect.New(t).Interface(), nil
        }
        return nil, &notRegisteredErr{kind: kind,
version: versionName}
    }
    return nil, &notRegisteredErr{kind: kind, version:

```

```
versionName}  
}
```

而pkg/conversion/encode.go与decode.go则在conversion.Scheme提供的基础功能之上，完成了最终的序列化、反序列化功能。下面是encode.go里的主方法EncodeToVersion (..) 的关键代码片段：

```
//确定要转换的源对象的版本号和类别  
objVersion, objKind, err :=  
s.ObjectVersionAndKind(obj) 象  
//生成目标版本的空对象  
objOut, err := s.NewObject(destVersion, objKind)  
//生成转换过程中所需的Metadata信息  
flags, meta := s.generateConvertMeta(objVersion,  
destVersion, obj)  
//调用converter的方法将源对象的数据填充到目标对象objOut  
err = s.converter.Convert(obj, objOut, flags, meta)  
//用JSON将目标对象转换成byte[]数组，完成序列化过程  
data, err = json.Marshal(obj)
```

再进一步，Kubernetes在conversion.Scheme的基础上又做了一个封装工具类runtime.Scheme，可以看作前者的代理类，主要增加了fieldLabelConversionFuncs这个Map属性，用于解决数据对象的属性名称的兼容性转换和校验，比如将需要兼容Pod的spec.host属性改为spec.nodeName的情况。

注意到`conversion.Scheme`只是实现了一个序列化与类型转换的框架API，提供了注册资源数据类型与转换函数的功能，那么具体的资源数据对象类型、转换函数又是在哪个包里实现的呢？答案是`pkg/api`。Kubernetes为不同的API版本提供了独立的数据类型和相关的转换函数并按照版本号命名Package，如`pkg/api/v1`、`pkg/api/v1beta3`等，而当前默认版本（内部版本）则存在于`pkg/api`目录下。

以`pkg/api/v1`为例，在每个目录里都包括如下关键源码：

- `types.go`定义了Rest API接口里所涉及的所有数据类型，`v1`版本有2000行代码；
- 在 `conversion.go` 与 `conversion_generated.go` 里 定义了 `conversion.Scheme`所需的从内部版本到`v1`版本的类型转换函数，其中`conversion_generated.go`中的代码有5000行之多，当然这是通过工具自动生成的代码；
- `register.go`负责将`types.go`里定义的数据类型与`conversion.go`里定义的数据转换函数注册到`runtime.Schema`里。

`pkg/api` 里的 `register.go` 初始化生成并持有一个全局的 `runtime.Scheme` 对象，并将当前默认版本的数据类型（`pkg/api/types.go`）注册进去，相关代码如下：

```
var Scheme = runtime.NewScheme()

func init() {
    Scheme.AddKnownTypes("",
        &Pod{},
        &PodList{},
        &PodStatusResult{},
```

```
        &PodTemplate{},
        &PodTemplateList{},
        &ReplicationControllerList{},
//此次省略30多个数据类型
        &ServiceList{},
        &Service{},
        &NodeList{},
        &Node{},
//省略
```

而pkg/api/v1/register.go与v1beta3下的register.go在初始化过程中分别把与版本相关的数据类型和转换函数注册到全局的runtime.Scheme中:

```
func init() {
    // Check if v1 is in the list of supported API
versions.
    if !registered.IsRegisteredAPIVersion("v1") {
        return
    }

    // Register the API.
    addKnownTypes()
    addConversionFuncs()
    addDefaultingFuncs()
}
```

这样一来，其他地方都可以通过`runtime.Scheme`这个全局变量来完成Kubernetes API中的数据对象的序列化和反序列化逻辑了，比如Kubernetes API Client包就大量使用了它，下面是`pkg/client/pods.go`里Pod删除的`Delete ()`方法的代码：

```
        // Delete takes the name of the pod, and returns an
error if one occurs

        func (c *pods) Delete(name string, options
*api.DeleteOptions) error {
            // TODO: to make this reusable in other client
libraries
            if options == nil {

                                                    return
c.r.Delete().Namespace(c.ns).Resource("pods").Name(name).Do
().Error()
            }
            body, err := api.Scheme.EncodeToVersion(options,
c.r.APIVersion())
            if err != nil {
                return err
            }

                                                    return
c.r.Delete().Namespace(c.ns).Resource("pods").Name(name).Bo
dy(body).Do().Error()
        }
```

清楚了Kubernetes Rest API中的数据对象的序列化机制及多版本的实现原理之后，我们接着分析下面这个重要流程的实现细节。

Kubernetes 中 实 现 了 `rest.Storage` 接 口 的 服 务 在 转 换 成 `restful.RouteFunction`以后，是怎样处理一个Rest请求并最终完成基于后端存储服务etcd上的具体操作过程的？

首先，Kubernetes 设计了一个名为“注册表”的Package（`pkg/registry`），这个Package按照`rest.Storage`服务所管理的资源数据的类型而划分为不同的子包，每个子包都由相同命名的一组Golang代码来完成具体的Rest接口的实现逻辑。

下面我们以Pod的Rest服务实现为例，其与“注册表”相关的代码位于`pkg/registry/pod`中，在`registry.go`里定义了Pod注册表服务的接口：

```
type Registry interface {
    // ListPods obtains a list of pods having labels
    which match selector.
    ListPods(ctx api.Context, label labels.Selector)
    (*api.PodList, error)
    // Watch for new/changed/deleted pods
    WatchPods(ctx api.Context, label labels.Selector,
    field fields.Selector, resourceVersion string)
    (watch.Interface, error)
    // Get a specific pod
    GetPod(ctx api.Context, podID string) (*api.Pod,
    error)
    // Create a pod based on a specification.
```

```
CreatePod(ctx api.Context, pod *api.Pod) error
// Update an existing pod
UpdatePod(ctx api.Context, pod *api.Pod) error
// Delete an existing pod
DeletePod(ctx api.Context, podID string) error
}
```

我们看到这个Pod注册表服务是针对Pod的CRUD的操作接口的一个定义，在入口参数中除了调用的上下文环境`api.Context`，就是我们之前分析过的`pkg/api`包中的Pod这个资源数据对象。为了实现强类型的方法调用，在`registry.go`里定义了一个名为`storage`的结构体，`storage`实现`Registry`接口，可以看作一种代理设计模式，因为具体的操作都是通过内部`rest.StandardStorage`来实现的。下面是截取的`registry.go`中的`create`、`update`、`delete`的源码：

```
func (s *storage) CreatePod(ctx api.Context, pod
*api.Pod) error {
    _, err := s.Create(ctx, pod)
    return err
}

func (s *storage) UpdatePod(ctx api.Context, pod
*api.Pod) error {
    _, _, err := s.Update(ctx, pod)
    return err
}
```

```

func (s *storage) DeletePod(ctx api.Context, podID
string) error {
    _, err := s.Delete(ctx, podID, nil)
    return err
}

```

那么，这个实现了`rest.StandardStorage`通用接口的真正`Storage`又是什么？从`Master`对象的初始化函数中，我们发现了下面的相关代码：

```

func (m *Master) init(c *Config) {
    healthzChecks := []healthz.HealthzChecker{}
    m.clock = util.RealClock{}
    podStorage := podetcd.NewStorage(c.EtcdHelper,
c.KubeletClient)
    podRegistry := pod.NewRegistry(podStorage.Pod)
}

```

`Master`对象创建了一个私有变量`podStorage`，其类型为`PodStorage`（`pkg/registry/pod/etcd/etcd.go`），`Pod`注册表服务实例（`podRegistry`）里真正的`Storage`是`podStorage.Pod`。下面是`podetcd`的函数`NewStorage`中的关键代码：

```

func NewStorage(h tools.EtcdHelper, k
client.ConnectionInfoGetter) PodStorage {
    store := &etcdgeneric.Etcd{
        NewFunc: func() runtime.Object { return
&api.Pod{} },
        NewListFunc: func() runtime.Object { return

```



```

&api.PodList{} },

.....

return PodStorage{
    Pod:          &REST{*store},
    Binding:      &BindingREST{store: store},
    Status:       &StatusREST{store: &statusStore},
    Log:          &LogREST{store: store,
kubeletConn: k},
    Proxy:        &ProxyREST{store: store},
    Exec:         &ExecREST{store: store,
kubeletConn: k},
    PortForward: &PortForwardREST{store: store,
kubeletConn: k},
}

```

在上述代码中我们看到：位于pkg/registry/generic/etcd/etcd.go里的etcd才是真正的Storage实现。而具体操作etcd的代码是靠tools.EtcdHelper这个类完成的，通过分析etcd.go里的func（e*Etcd）Create（ctx api.Context，obj runtime.Object）方法，我们知道创建一个etcd里的键值对的关键逻辑如下。

- 获取对象的名字：name, err: =e.ObjectNameFunc（obj）。
- 获取Key: key, err: =e.KeyFunc（ctx, name）。
- 生成一个空的Object对象：out: =e.NewFunc（）。
- 将键值对写入etcd: 在e.Helper.CreateObj（key, obj, out, ttl）方法中通过调用runtime.Codec完成从对象到字符串的转换，最终保存到etcd中。

- 回调创建完成后的处理逻辑：e.AfterCreate（out）。

注意到之前PodStorage创建store时重载了ObjectNameFunc（）、KeyFunc（）、NewFunc（）等函数，于是完成了针对Pod的创建过程，Kubernetes API服务中的其他数据对象也都遵循同样的设计模式。

进一步研究代码，我们发现PodStorage中的Pod、Binding、Status等属性是pkg/api/rest/rest.go中几个不同的Rest接口的实现，并且通过etcdgeneric.Etcd这个实例来完成Pod的一些具体操作，比如这里的StatusREST。下面是其相关代码片段：

```
// StatusREST implements the REST endpoint for
changing the status of a pod.

type StatusREST struct {
    store *etcdgeneric.Etcd
}

// New creates a new pod resource
func (r *StatusREST) New() runtime.Object {
    return &api.Pod{}
}

// Update alters the status subset of an object.
func (r *StatusREST) Update(ctx api.Context, obj
runtime.Object) (runtime.Object, bool, error) {
    return r.store.Update(ctx, obj)
}
```

表 6.2 展现了 PodStorage 中的各个 XXXREST 接口与 pkg/api/rest/rest.go 里的相关 Rest 接口的一一对应关系。

表6.2 PodStorage中的各个XXXREST接口与
pkg/api/rest/rest.go里的相关Rest接口的一一对应关系

PodStorage Rest 接口	对应 API Rest 框架的接口	接 口 功 能
REST	rest.Redirector rest.CreatorUpdater rest.Lister rest.Watcher rest.GracefulDeleter rest.Getter	重定向资源的路径 资源创建、更新接口 资源列表查询接口 Watcher 资源变化接口 支持延迟的资源删除接口 获取具体资源的信息接口
BindingREST	rest.Creator	创建资源的接口
StatusREST	Rest.Updater	更新资源的接口
LogREST	rest.GetterWithOptions	获取资源的接口
ExecREST\ProxyREST\PortForwardREST	rest.Connector	连接资源的接口

其中PodStorage.REST接口究竟实现了哪些API Rest接口，这个比较隐晦，笔者也花费了一些时间来研究这个问题，这涉及Go语言的一个特殊特性：结构体内嵌一个其他类型的结构体指针，就可以使用内嵌结构体的方法，相当于面向对象语言中的“继承”。而PodStorage.REST恰恰嵌套了etcdgeneric.Etcd类型的匿名指针：`&REST{*store}`，而etcdgeneric.Etcd则实现了rest.Creator、rest.Lister、rest.Watcher等资源管理接口的所有方法，PodStorage.REST也“继承”了这些接口。

我们回头看看下面这段来自api_installer.go的registerResourceHandlers函数中的片段：

```

    creator, isCreator := storage.(rest.Creator)
        namedCreator, isNamedCreator := storage.
(named.NamedCreator)
    lister, isLister := storage.(rest.Lister)
    getter, isGetter := storage.(rest.Getter)

```

```

        getterWithOptions, isGetterWithOptions := storage.
(rest.GetterWithOptions)
        deleter, isDeleter := storage.(rest.Deleter)
        gracefulDeleter, isGracefulDeleter := storage.
(rest.GracefulDeleter)
        updater, isUpdater := storage.(rest.Updater)
        patcher, isPatcher := storage.(rest.Patcher)
        watcher, isWatcher := storage.(rest.Watcher)
        _, isRedirector := storage.(rest.Redirector)
        connector, isConnector := storage.(rest.Connector)
        storageMeta, isMetadata := storage.
(rest.StorageMetadata)

```

上述代码对 `storage` 对象进行判断，以确定并标记它所满足的 `APIRest` 接口类型，而接下来的这段代码在此基础上确定此接口所包含的 `actions`，后者则对应到某种 `HTTP` 请求方法（`GET/POST/PUT/DELETE`）或者 `HTTP PROXY`、`WATCH`、`CONNECT` 等动作：

```

        ctions = appendIf(actions, action{"GET", itemPath,
nameParams, namer}, isGetter)
        actions = appendIf(actions, action{"PATCH", itemPath,
nameParams, namer}, isPatcher)
        actions = appendIf(actions, action{"DELETE", itemPath,
nameParams, namer}, isDeleter)
        actions = appendIf(actions, action{"WATCH", "watch/" +
itemPath, nameParams, namer}, isWatcher)

```

```

        actions = appendIf(actions, action{"PROXY", "proxy/" +
itemPath +("/{path:*}", proxyParams, namer}, isRedirector)
        actions = appendIf(actions, action{"CONNECT",
itemPath, nameParams, namer}, isConnector)

```

我们注意到rest.Redirector类型的storage被当作PROXY进行处理，由 apiserver.ProxyHandler 进行拦截，并调用 rest.Redirector 的 ResourceLocation方法获取到资源的处理路径（可能包括一个非空的 http.RoundTripper，用于处理执行 Redirector 返回的 URL 请求）。Kubernetes API Server中PROXY请求存在的意义在于透明地访问其他某个节点（比如某个Minion）上的API。

最后，我们来分析下registerResourceHandlers中完成从rest.Storage到restful.Route映射的最后一段关键代码。下面是rest.Getter接口的Storage的映射代码：

```

        case "GET": // Get a resource.
            var handler restful.RouteFunction
            handler = GetResource(getter, reqScope)
            doc := "read the specified " + kind

                                                    route :=
ws.GET(action.Path).To(handler).Filter(m).Doc(doc).
        Param(ws.QueryParameter("pretty", "If 'true', then the
output is pretty printed.")).

Operation("read"+namespaced+kind+strings.Title(subresource)
).

```

```

Produce(append(storageMeta.ProducesMIMETypes(action.Verb),
"application/json"...)).
Returns(http.StatusOK, "OK",
versionedObject).Writes(versionedObject)

addParams(route, action.Params)
ws.Route(route)

```

上述代码首先通过函数 `GetResource()` 创建了一个 `restful.RouteFunction`，然后生成一个 `restful.route` 对象，最后注册到 `restful.WebService` 中，从而完成了 `rest.Storage` 到 `Rest` 服务的“最后一公里”通车。`GetResource()` 函数存在于 `pkg/apiserver/resthandler.go` 里，`resthandler.go` 提供了各种具体的 `restful.RouteFunction` 的实现函数，是真正触发 `rest.Storage` 调用的地方。下面是 `GetResource()` 方法的主要代码，可以看出这里是调用 `rest.Getter` 接口的 `Get()` 方法以返回某个资源对象：

```

func GetResource(r rest.Getter, scope RequestScope)
restful.RouteFunction {
    return getResourceHandler(scope,
        func(ctx api.Context, name string, req
        *restful.Request) (runtime.Object, error) {
            return r.Get(ctx, name)
        })
}

```

看了上面的代码，你可能会有一个疑问：“说好的权限控制呢？”别急，看看下面的资源创建的`createHandler()` 代码：

```
    if admit.Handles(admission.Create) {
        userInfo, _ := api.UserFrom(ctx)

        err =
admit.Admit(admission.NewAttributesRecord(obj,    scope.Kind,
namespace,    name,    scope.Resource,    scope.Subresource,
admission.Create, userInfo))

        if err != nil {
            errorJSON(err, scope.Codec, w)
            return
        }
    }
}
```

资源的Update、Delete、Connect、Patch等操作都有类似的权限控制，从Admit的参数`admission.Attributes`的属性来看，第三方系统可以开发细粒度的权限控制插件，针对任意资源的任意属性进行细粒度的权限控制，因为资源对象本身都传递到参数中了。

对Kubernetes Rest API Server的复杂实现机制和调用流程的总结如下。

- 在`pkg/api`包里定义了Rest API中涉及的资源对象、提供的Rest接口、类型转换框架和具体转换函数、序列化反序列化等代码。其中，资源对象和转换函数按照版本分包，形成了Kubernetes API Server基础的框架，其中核心是各类资源（如Node、Pod、

PodTemplate 、 Service 等） 及这些资源对应的rest.Storage （Rest API接口） 。

- 在pkg/runtime包里最重要的对象是Schema， 它保存了Kubernetes API Service中注册的资源对象类型、转换函数等重要基础数据。另外， runtime包也提供了获取json/yaml序列化、反序列化的Codec结构体， runtime总体上与pkg/api密切关联， 分离出来的目的是供其他模块方便使用。
- pkg/registry包其实是把pkg/api中定义的各种资源对象所提供的Rest 接口进一步规范定义并且实现对应的接口， 其中generate/etcd/etcd.go里的etcd对象是一个真正实现了rest.Storage接口的基于etcd后端存储的服务框架， 并且Kubernetes中的各种资源对象的具体Storage实现也是通过它来完成真正的“后端存储操作”。
- Kubernetes采用了go-restful这个第三方的Rest框架， 大大简化了Rest服务的开发， 主要代码在pkg/apiserver源码包里。通过APIGroupVersion这个结构体可完成不同API版本的Rest路径映射， 而api_installer.go则实现了从Kubernetes Rest.Storage接口到go-restful 的映射连接逻辑， 对应 rest.Storage 的具体restful.RouteFunction则在resthandler.go里实现。

6.2.3 设计总结

如果你耐心看完了上面的每一段文字和代码，而且尝试追踪源码来加深对6.2.1节内容的理解，那么笔者相信你对于Kubernetes API Server的设计的第一个评价就是：“太复杂、太反常了！不就是一个Rest Server么，如果用Java语言，我可以分分钟搞定一个！”当然，你肯定有以下或者更多的假设。

- 放弃多版本API的兼容需求。
- 只采用一个特定的后端存储实现。
- API只接收一种输入输出格式，比如JSON或者YAML，而不是两种或更多。
- 放弃Watch这种高难度的API。
- 不实现Proxy代理。
- 不做可拔插的权限控制设计（或者根本没有）。
- 每新增一种资源类型，就从头写很多代码来实现该资源的Rest服务。

虽然代码很复杂，但我们不得不承认，Kubernetes API Server是一个精心“设计”的系统。

什么样的设计是一个好的设计？这个问题没有标准答案，但有一点是大家都认可的：好的设计要尽量提供一种好的框架机制，方便未来增加新功能或者自定义扩展某些特性。我们以这个标准对

Kubernetes API Server的设计进行评价，就会发现：它的设计真的很好。

我们先分析一下Kubernetes API Server的“领域模型”。API Server里的Rest服务都是针对某个“资源对象”的操作，这些操作可以分为新增、修改、列表输出、删除、Watch变化、代理请求及连接资源等基础操作，大多数操作都是与后端存储的交互。因为只是基本的资源数据对象的增、删、改、查，所以主体逻辑是通用的，比如序列化、反序列化、基于Key-Value的存储，以及这个过程中的数据校验和权限控制等问题。

通过以上分析，我们发现这个系统的核心对象只有两个：资源对象与操作资源对象的Storage服务。虽然各个资源的Storage服务的主体功能相同，都是将资源存储到etcd这个Key-Value后端存储系统上并提供相关操作，但不同类型资源的Storage服务的接口和具体逻辑还是有差别的，比如某类资源是不允许更新的，有些资源则允许“Connect”，所以这里的设计是Kubernetes API Server的最有代表性的经典设计——资源服务接口的细分与组合设计。

如图6.2所示是此设计的全景图（以Pod资源对象为例）。资源服务接口被拆分为rest.Create、rest.Updater、rest.CreateUpdate（组合了Create与Updater接口）、rest.GracefulDelete（支持延迟删除资源的接口）、rest.Patcher（组合更新与Get接口）、rest.Connect（开启HTTP连接到该资源进行操作，比如连接到一个Pod中执行某个bash命令）等10个细分接口。

考虑到大多数资源对象都需要基本的CRUD接口，这就是rest.StandardStorage这个聚合型“标准存储服务”接口出现的原因。而作

为StandardStorage的默认实现，pkg/registry/generic/etcd/etcd.go里etcd这个对象实现了基于etcd后端存储的所有具体操作，而各种资源的Storage服务则通过将请求代理到etcd对象上来完成具体的功能。

这里有点让人难以理解的是PodStorage与它的属性Pod的关系，其实PodStorage这个对象是一个聚合了与Pod相关的各个资源的存储服务，多看一下它的定义就能立刻明白了：

```
// PodStorage includes storage for pods and all sub
resources

type PodStorage struct {
    Pod          *REST
    Binding      *BindingREST
    Status       *StatusREST
    Log          *LogREST
    Proxy        *ProxyREST
    Exec         *ExecREST
    PortForward  *PortForwardREST
}
```

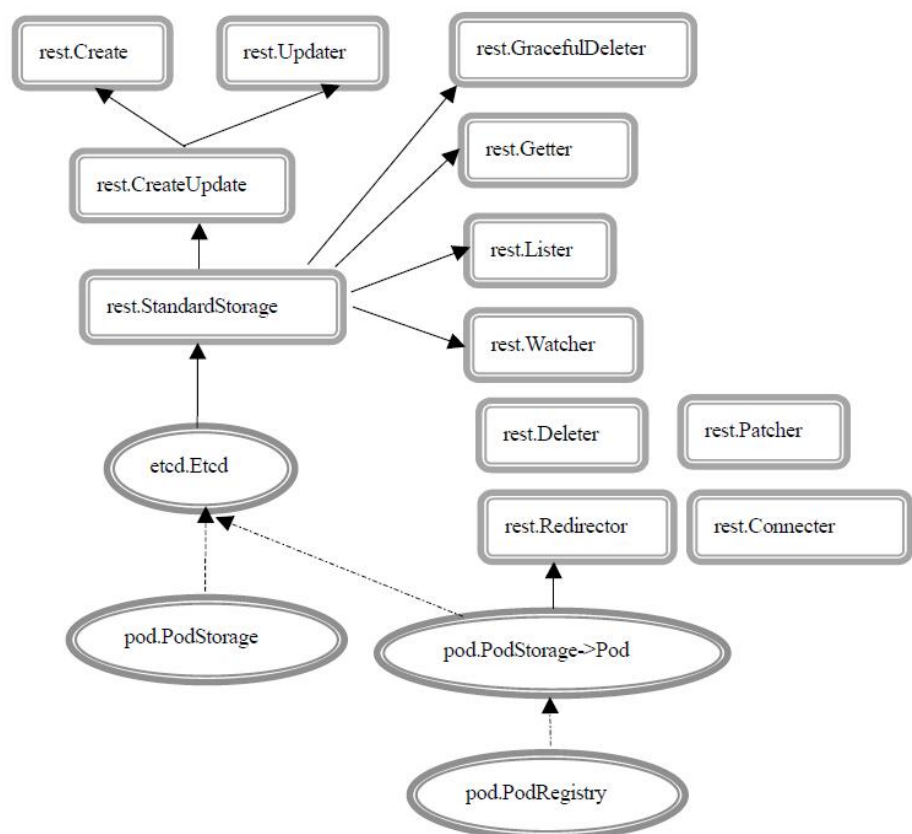


图6.2 API Server的Storage设计全景图

所以，这里的PodStorage应该重命名为AllPodResStorage，而真正的PodStorage上就是里面的那个Pod变量，这个变量是对etcd实例的一个引用，然后又实现了rest.Redirector接口。现在你终于能理解PodRegistry引用Pod变量而不是PodStorage来实现Pod操作的真正原因了吧？

最后，我们来说说PodRegistry存在的目的。从之前的代码分析来看，一个来自外部的针对某个资源的Rest API发起的请求最后落到对应资源的rest.Storage对象上，由restful.RouteFunction调用此对象的相关方法完成资源的操作并生成应答返回给客户端，这个过程并没有涉及对应资源的Registry服务。那么问题来了，资源的Registry接口存在的

理由是什么呢？答案很简单，对比Storage接口与Registry中的资源创建方法的签名，下面是二者的源码对比，后者更符合“手工调用”：

Storage中创建通用的资源对象的接口

```
Create(ctx api.Context, obj runtime.Object)
(runtime.Object, error)
```

PodRegistry中创建Pod资源的接口

```
CreatePod(ctx api.Context, pod *api.Pod) error
```

在Kubernetes API Server中为每类资源都创建并提供了一个Registry接口服务的目的是供内部模块的编程使用，而非对外提供服务，很多文档都错误理解了这个问题。

本节最后给出了如图6.3所示的经典的Kubernetes的Master节点数据流图，此刻这个图在你眼里可能已经什么都不算了，因为你已经洞穿了幕后的一切。

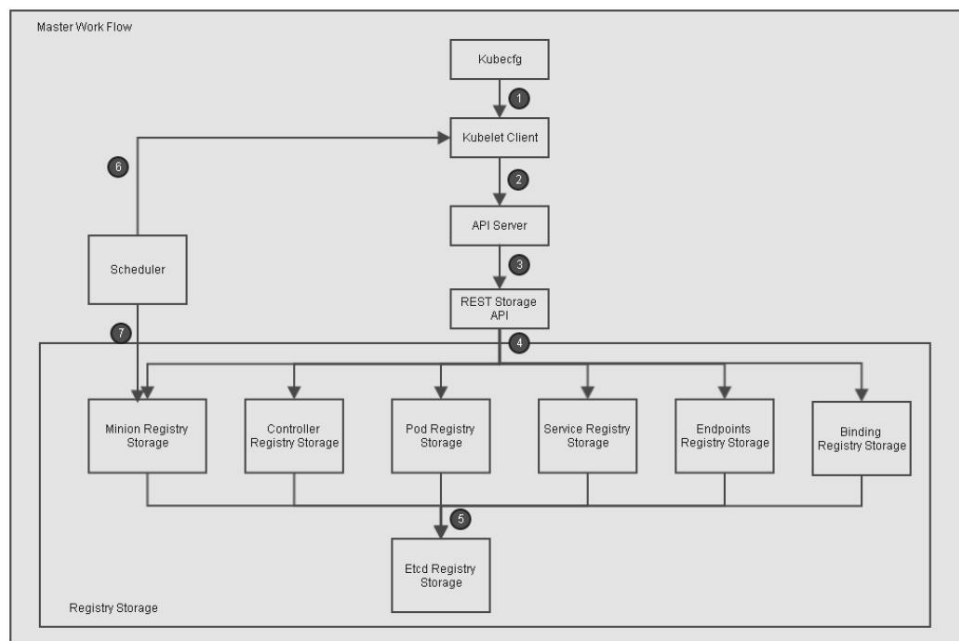


图6.3 Master节点数据流图

6.3 kube-controller-manager进程源码分析

运行在Master节点上的第2个进程就是kube-controller-manager进程，即Controller Manager Server，Kubernetes的核心进程之一，其主要目的是实现Kubernetes集群的故障检测和恢复的自动化工作，比如内部组件EndpointController控制器负责Endpoints对象的创建和更新；ReplicationManager根据注册表中的ReplicationController的定义，完成Pod的复制或者移除，以确保复制数量的一致性；NodeController负责Minion节点的发现、管理和监控。

6.3.1 进程启动过程

kube-controller-manager进程的入口源码位置如下:

```
github.com/GoogleCloudPlatform/kubernetes/cmd/kube-  
controller-manager/ controller-manager.go
```

入口main () 函数的逻辑如下:

```
func main() {  
    runtime.GOMAXPROCS(runtime.NumCPU())  
    s := app.NewCMServer()  
    s.AddFlags(pflag.CommandLine)  
    util.InitFlags()  
    util.InitLogs()  
    defer util.FlushLogs()  
    verflag.PrintAndExitIfRequested()  
    if err := s.Run(pflag.CommandLine.Args()); err !=  
nil {  
        fmt.Fprintf(os.Stderr, "%v\n", err)  
        os.Exit(1)  
    }  
}
```

从源码可以看出，关键代码只有两行，创建一个CMServer并调用Run方法启动服务。下面我们分析CMServer这个结构体，它是Controller Manager Server进程的主要上下文数据结构，存放一些关键参数，表6.3是对CMServer里的关键参数的解释。

表6.3 CMServer的重要属性

属 性 名	默 认	含 义
ConcurrentEndpointSyncs	5 秒	并发执行的 Endpoint 的同步任务的数量
ConcurrentRCSyncs	5 秒	并发执行的 Replication Controller 的同步任务的数量
NodeSyncPeriod	5 秒	从 CloudProvider 处同步 Node 节点的周期
NodeMonitorPeriod	5 秒	Node 节点监控的周期
ResourceQuotaSyncPeriod	10 秒	对资源的配额使用情况进行同步的周期
NamespaceSyncPeriod	5 分钟	Namespace 同步的周期
PVClaimBinderSyncPeriod	10 秒	对 PV（持久存储）和 PV 的申请进行同步的周期
PodEvictionTimeout	5 分钟	在 Node 失败的情况下，其上的 Pod 多久后才被删除
master		Kubernetes API Server 的访问地址

从上述这些变量来看，Controller Manager Server其实就是一个“超级调度中心”，它负责定期同步Node节点状态、资源使用配额信息、Replication Controller、Namespace、Pod的PV绑定等信息，也包括执行诸如监控Node节点状态、清除失败的Pod容器记录等一系列定时任务。

在controller-manager.go里创建CMServer实例并把参数从命令行中传递到CMServer后，就调用它的func (s*CMServer) Run ([]string)方法进入关键流程，这里首先创建一个Rest Client对象用于访问Kubernetes API Server提供的API服务：

```

    kubeClient, err := client.New(kubeconfig)
    if err != nil {
        glog.Fatalf("Invalid API configuration: %v",
```

```
err)
```

```
}
```

随后，创建一个HTTP Server以提供必要的性能分析（Performance Profile）和性能指标度量（Metrics）的Rest服务：

```
go func() {
    mux := http.NewServeMux()
    healthz.InstallHandler(mux)
    if s.EnableProfiling {
        mux.HandleFunc("/debug/pprof/",
pprof.Index)
        mux.HandleFunc("/debug/pprof/profile",
pprof.Profile)
        mux.HandleFunc("/debug/pprof/symbol",
pprof.Symbol)
    }
    mux.Handle("/metrics", prometheus.Handler())

    server := &http.Server{
                                                Addr:
net.JoinHostPort(s.Address.String(), strconv.Itoa(s.Port)),
        Handler: mux,
    }
    glog.Fatal(server.ListenAndServe())
}()
```

我们注意到性能分析的Rest路径是以/debug开头的，表明是为了程序调试所用，事实上的确如此，这里的几个Profile选项都是针对当前Go进程的Profile数据，比如我们在Master节点上执行curl命令（地址为http://127.0.0.1: 10252/debug/pprof/heap）可以获取进程的当前堆栈信息，会输出如下信息：

```
heap profile: 4: 78112 [1109: 824584] @ heap/1048576
      1: 32768 [1: 32768] @ 0x402612 0x75ab95 0x771419
0x771379 0x565f08 0x46133f 0x400d10 0x4155a3 0x43e711
      1: 32768 [1: 32768] @ 0x408806 0x407968 0x97e591
0x9895aa 0x76099b 0xa2f400 0xa4e887 0x765dc4 0x557fbc
0x782fac 0x5fe5db 0x602ca7 0x462c92 0x400f06 0x415594
0x43e711
      1: 12288 [1: 12288] @ 0x4199fc 0x7df75d 0x5b585c
0x5b4947 0x5b405a 0x5aa472 0x5aa2b7 0x5aa188 0x5ad0d3
0x46291e 0x43e711
      1: 288 [1: 288] @ 0x415d6a 0x43276f 0x43510f 0x42fd37
0x4311f9 0x430ef5 0x43c136
```

其他还有GC回收、Symbol查看、进程30秒内的CPU利用率、协程的阻塞状态等Profile功能，输出的数据格式符合google-perf-tools这个工具的要求，因此可以做运行期的可视化Profile，以便排查当前进程潜在的问题或性能瓶颈。

性能指标度量目前主要收集和统计Kubernetes API Server的Rest API的调用情况，执行curl（http://127.0.0.1: 10252/metrics），可以看到输出中包括大量类似下面的内容：

```
rest_client_request_latency_microseconds{url="http://centos
-
master:8080/api/v1/namespaces/default/endpoints/%3Cname%3E"
,verb="GET",quantile="0.5"} 1448

rest_client_request_latency_microseconds{url="http://centos
-
master:8080/api/v1/namespaces/default/endpoints/%3Cname%3E"
,verb="GET",quantile="0.9"} 1699

rest_client_request_latency_microseconds{url="http://centos
-
master:8080/api/v1/namespaces/default/endpoints/%3Cname%3E"
,verb="GET",quantile="0.99"} 2093
```

这些指标有助于协助发现Controller Manager Server在调度方面的性能瓶颈，因此可以理解为什么会被包括到进程代码中去。

接下来，启动流程进入到关键代码部分。在这里，启动进程分别创建如下控制器，这些控制器的主要目的是实现资源在Kubernetes API Server的注册表中的周期性同步工作：

- EndointController负责对注册表中的Kubernetes Service的Endpoints信息的同步工作；
- ReplicationManager根据注册表中对ReplicationController的定义，完成Pod的复制或者移除，以确保复制数量的一致性；

- NodeController则通过CloudProvider的接口完成Node实例的同步工作；
- servicecontroller通过CloudProvider的接口完成云平台中的服务的同步工作，这些服务目前主要是外部的负载均衡服务；
- ResourceQuotaManager负责资源配额使用情况的同步工作；
- NamespaceManager负责Namespace的同步工作；
- PersistentVolumeClaimBinder 与 PersistentVolumeRecycler 分别完成 PersistentVolume的绑定和回收工作；
- TokensController、ServiceAccountsController分别完成Kubernetes服务的Token、Account的同步工作。

创建并启动完成上述的控制器以后，各个控制器就开始独立工作，Controller Manager Server启动完毕。

6.3.2 关键代码分析

在6.3.1节对kube-controller-manager进程的启动过程进行了详细分析，我们发现这个进程的主要逻辑就是启动一系列的“控制器”。这里以Kubernetes里比较关键的Pod副本（Pod Replica）数量的控制实现过程为例，来分析完成这个任务的“控制器”——ReplicationManager具体是如何工作的。

首先，我们来看看ReplicationManager结构体的定义：

```
type ReplicationManager struct {
    kubeClient client.Interface
    podControl PodControlInterface

    // An rc is temporarily suspended after
    creating/deleting these many replicas.
    // It resumes normal action after observing the
    watch events for them.
    burstReplicas int
    // To allow injection of syncReplicationController
    for testing.
    syncHandler func(rcKey string) error

    // podStoreSynced returns true if the pod store
    has been synced at least once.
```

```

        // Added as a member to the struct to allow
injection for testing.
        podStoreSynced func() bool

        // A TTLCache of pod creates/deletes each rc
expects to see
        expectations RCExpectationsManager
        // A store of controllers, populated by the
rcController
        controllerStore cache.StoreToControllerLister
        // A store of pods, populated by the podController
        podStore cache.StoreToPodLister
        // Watches changes to all replication controllers
        rcController *framework.Controller
        // Watches changes to all pods
        podController *framework.Controller
        // Controllers that need to be updated
        queue *workqueue.Type
    }

```

在上述结构体里，比较关键的几个属性如下。

- **kubeClient**: 用来访问KubernetesAPIServer的Rest客户端，这里用来访问注册表中定义的ReplicationController对象并操作Pod。
- **podControl**: 实现了Pod副本创建的函数，其实现类为RealPodControl（位于kubernetes/pkg/controller/controller_utils.go）。

- `syncHandler`: 是RC (`ReplicationController`) 的同步实现方法, 完成具体的RC同步逻辑 (创建Pod副本时调用PodControl的相关方法) , 在代码中其被赋值给 `ReplicationManager.syncReplicationController` 方法。
- `expectations`: 是Pod副本在创建、删除过程中的流控机制的重要组成部分。
- `controllerStore`: 是一个具备本地缓存功能的通用的资源存储服务, 这里存放 `framework.Controller` 运行过程中从Kubernetes API Server同步过来的资源数据, 目的是减轻资源同步过程中对Kubernetes API Server造成的访问压力并提高资源同步的效率。
- `rcController`: `framework.Controller` 的一个实例, 用来实现RC同步的任务调度逻辑。
- `framework.Controller`: 是 `kube-controller-manager` 里设计的用于资源对象同步逻辑的专用任务调度框架。
- `podStore`: 类似于 `controllerStore` 的作用, 用来存取和获取Pod资源对象。
- `podController`: 类似于 `rcController` 的作用, 用来实现Pod同步的任务调度逻辑。

理解了 `ReplicationManager` 结构体的重要参数及其作用之后, 我们来看 `controller.NewReplicationManager (kubeClient client.Interface , burstReplicas int) *ReplicationManager` 这个构造函数中的关键代码, 注意到这里通过调用 `framework.NewInformer ()` 方法先后创建了用于RC同步及Pod同步的 `framework.Controller`。下面是 `framework.NewInformer ()` 方法的源码:

```

func NewInformer(
    lw cache.ListerWatcher,
    objType runtime.Object,
    resyncPeriod time.Duration,
    h ResourceEventHandler,
) (cache.Store, *Controller) {
    clientState :=
cache.NewStore(DeletionHandlingMetaNamespaceKeyFunc)
    fifo :=
cache.NewDeltaFIFO(cache.MetaNamespaceKeyFunc, nil,
clientState)
    cfg := &Config{
        Queue:          fifo,
        ListerWatcher:   lw,
        ObjectType:      objType,
        FullResyncPeriod: resyncPeriod,
        RetryOnError:    false,
        Process: func(obj interface{}) error {
            // from oldest to newest
            for _, d := range obj.(cache.Deltas) {
                switch d.Type {
                    case cache.Sync, cache.Added,
cache.Updated:
                        if old, exists, err :=
clientState.Get(d.Object); err == nil && exists {
                            if err :=

```

```

        clientState.Update(d.Object); err != nil {
            return err
        }
        h.OnUpdate(old, d.Object)
    } else {
        if err :=
        clientState.Add(d.Object); err != nil {
            return err
        }
        h.OnAdd(d.Object)
    }
    case cache.Deleted:
        if err :=
        clientState.Delete(d.Object); err != nil {
            return err
        }
        h.OnDelete(d.Object)
    }
    return nil
},
}
return clientState, New(cfg)
}

```

在上述代码中，lw（ListerWatcher）用来获取和监测资源对象的变化，而fifo则是一个DeltaFIFO的Queue，用来存放变化的资源（需要

同步的资源)。当Controller框架发现有变化的资源需要处理时，就会将新资源与本地缓存clientState中的资源进行对比，然后调用相应的资源处理函数ResourceEventHandler的方法，完成具体的处理逻辑。下面是针对RC的ResourceEventHandler的具体实现：

```
framework.ResourceEventHandlerFuncs{
    AddFunc: rm.enqueueController,
    UpdateFunc: func(old, cur interface{}) {
        oldRC := old.
(*api.ReplicationController)
        curRC := cur.
(*api.ReplicationController)
        if oldRC.Status.Replicas !=
curRC.Status.Replicas {
            glog.V(4).Infof("Observed updated
replica count for rc: %v, %d->%d", curRC.Name,
oldRC.Status.Replicas, curRC.Status.Replicas)
        }
        rm.enqueueController(cur)
    },
    DeleteFunc: rm.enqueueController,
}
```

在上述源码中，我们看到当RC里Pod的副本数量属性发生变化以后，ResourceEventHandler就将此RC放入ReplicationManager的queue队列中等待处理，为什么没有在这个handler函数中直接处理而是先放入队列再异步处理呢？最主要的一个原因是Pod副本创建的过程比较耗

时。Controller框架把需要同步的RC对象放入queue以后，接下来是谁在“消费”这个队列呢？答案就在ReplicationManager的Run（）方法中：

```
func (rm *ReplicationManager) Run(workers int, stopCh
<-chan struct{}) {
    defer util.HandleCrash()
    go rm.rcController.Run(stopCh)
    go rm.podController.Run(stopCh)
    for i := 0; i < workers; i++ {
        go util.Until(rm.worker, time.Second, stopCh)
    }
    <-stopCh
    glog.Infof("Shutting down RC Manager")
    rm.queue.ShutDown()
}
```

上述代码首先启动rcController与podController这两个Controller，启动之后，这两个Controller就分别开始拉取RC与Pod的变动信息，随后又启动N个协程并发处理RC的队列，其中func Until（f func（），period time.Duration，stopCh<-chan struct{}）方法的逻辑是按照指定的周期period执行方法f。下面是ReplicationManager的worker方法的源码，负责从RC队列中拉取RC并调用rm的syncHandler方法完成具体处理：

```
func (rm *ReplicationManager) worker() {
    for {
```

```

func() {
    key, quit := rm.queue.Get()
    if quit {
        return
    }
    defer rm.queue.Done(key)
    err := rm.syncHandler(key.(string))
    if err != nil {
        glog.Errorf("Error syncing replication
controller: %v", err)
    }
}()
}

```

从ReplicationManager的构造函数中我们得知：syncHandler在这里其实是func (rm*ReplicationManager) syncReplicationController (key string) 方法。下面是该方法的源码：

```

func (rm *ReplicationManager)
syncReplicationController(key string) error {
    startTime := time.Now()
    defer func() {
        glog.V(4).Infof("Finished syncing controller
%q (%v)", key, time.Now().Sub(startTime))
    }()

```

```

                                obj,    exists,    err    :=
rm.controllerStore.Store.GetByKey(key)
    if !exists {
        glog.Infof("Replication Controller has been
deleted %v", key)
        rm.expectations.DeleteExpectations(key)
        return nil
    }
    if err != nil {
        glog.Infof("Unable to retrieve rc %v from
store: %v", key, err)
        rm.queue.Add(key)
        return err
    }
    controller := *obj.(*api.ReplicationController)
    if !rm.podStoreSynced() {
        // Sleep so we give the pod reflector
goroutine a chance to run.
        time.Sleep(PodStoreSyncedPollPeriod)
        glog.Infof("Waiting for pods controller to
sync, requeuing rc %v", controller.Name)
        rm.enqueueController(&controller)
        return nil
    }

```

```

                                rcNeedsSync    :=
rm.expectations.SatisfiedExpectations(&controller)

```

```

                                podList,    err    :=
rm.podStore.Pods(controller.Namespace).List(labels.Set(cont
roller.Spec.Selector).AsSelector())
    if err != nil {
        glog.Errorf("Error getting pods for rc %q:
%v", key, err)
        rm.queue.Add(key)
        return err
    }

    filteredPods := filterActivePods(podList.Items)
    if rcNeedsSync {
        rm.manageReplicas(filteredPods, &controller)
    }

                                if    err    :=
updateReplicaCount(rm.kubeClient.ReplicationControllers(con
troller.Namespace), controller, len(filteredPods)); err !=
nil {
        rm.enqueueController(&controller)
    }
    return nil
}

```

在上述代码里有一个重要的流控变量`rcNeedsSync`。为了限流，在RC同步逻辑的过程中，一个RC每次最多执行N个Pod的创建、删除，如果某个RC的同步过程涉及的Pod副本数量超过`burstReplicas`这个阈

值，就会采用RCExpectations机制进行限流。RCExpectations对象可以理解为一个简单的规则：即在限定的时间内执行N次操作，每次操作都使计数器减一，计数器为零表示N个操作已经完成，可以进行下一批次的操作了。

Kubernetes为什么会设计这样一个流程控制机制？其实答案很简单——为了公平。因为谷歌的开发Kubernetes的资深大牛们早已预见到某个RC的Pod副本一次扩容至100倍的极端情况可能真实发生，如果没有流控机制，则这个巨无霸的RC同步操作会导致其他众多“散户”崩溃！这绝对不是谷歌的理念。

接着看上述代码里所调用的ReplicationManager的manageReplicas方法，这是RC同步的具体逻辑实现，此方法采用了并发调用的方式执行批量的Pod副本操作任务，相关代码如下：

```
wait := sync.WaitGroup{}
wait.Add(diff)
glog.V(2).Infof("Too few %q/%q replicas, need
%d, creating %d", controller.Namespace, controller.Name,
controller.Spec.Replicas, diff)
for i := 0; i < diff; i++ {
    go func() {
        defer wait.Done()
        if err :=
rm.podControl.createReplica(controller.Namespace,
controller); err != nil {
            glog.V(2).Infof("Failed creation,
```



```

    decrementing    expectations    for    controller    %q/%q",
    controller.Namespace, controller.Name)

    rm.expectations.CreationObserved(controller)
                                util.HandleError(err)
    }
    }()
}
wait.Wait()

```

追踪至此，我们才看到创建 Pod 副本的真正代码在 `PodControl.createReplica()` 方法里，而此方法的具体实现方法则是 `RealPodControl.createReplica()`，位于 `controller_utils.go` 里。通过分析该方法，我们可以知道创建 Pod 副本的过程就是创建一个 Pod 资源对象，并把 RC 中定义的 Pod 模板赋值给该 Pod 对象，并且 Pod 的名字用 RC 的名字做前缀，最后调用 Kubernetes Client 将 Pod 对象通过 Kubernetes API Server 写入后端的 etcd 存储中。

在本节最后，我们来分析一下 Controller 框架中如何实现资源对象的查询和监听逻辑并且在资源发生变动时回调 `Controller.Config` 对象中的 `Process` 方法：`func (obj interface{})`，最终完成整个 Controller 框架的闭环过程。

首先，在 Controller 框架中构建了 `Reflector` 对象以实现资源对象的查询和监听逻辑，它的源码位于 `pkg/client/cache/reflector.go` 中，我们看一下这个对象的数据结构就基本明白了其工作原理：

// Reflector watches a specified resource and causes all changes to be reflected in the given store.

```
type Reflector struct {  
    // The type of object we expect to place in the  
store.  
    expectedType reflect.Type  
    // The destination to sync up with the watch  
source  
    store Store  
    // listerWatcher is used to perform lists and  
watches.  
    listerWatcher ListerWatcher  
    // period controls timing between one watch ending  
and  
    // the beginning of the next one.  
    period time.Duration  
    resyncPeriod time.Duration  
    // lastSyncResourceVersion is the resource version  
token last  
    // observed when doing a sync with the underlying  
store  
    // it is thread safe, but not synchronized with  
the underlying store  
    lastSyncResourceVersion string  
    // lastSyncResourceVersionMutex guards read/write  
access to lastSyncResourceVersion
```

```
        lastSyncResourceVersionMutex sync.RWMutex
    }
}
```

核心思路就是通过`listerWatcher`去获取资源列表并监听资源的变化，然后存储到`store`中。这里你可能有个疑问，这个`store`究竟是哪个对象？是 `ReplicationManager` 里的 `controllerStore` 还是 `framework.NewInformer()` 方法里创建的`fifo`队列？

下面的两段来自`pkg/controller/framework/controller.go`的代码会告诉我们答案。

首先是来自 `Controller` 的 `run` 方法 `func (c*Controller) Run(stopCh<-chan struct{})` 的代码片段：

```
    r := cache.NewReflector(
        c.config.ListerWatcher,
        c.config.ObjectType,
        c.config.Queue,
        c.config.FullResyncPeriod,
    )
```

然后是来自 `Controller` 的 `NewInformer` 方法 `func NewInformer (lw cache.ListerWatcher , objType runtime.Object , resyncPeriod time.Duration , h ResourceEventHandler ,) (cache.Store , *Controller)` 中的代码片段：

```
    cfg := &Config{
        Queue:          fifo,
```

```
    ListerWatcher:    lw,  
    ObjectType:       objType,  
    FullResyncPeriod: resyncPeriod,  
    RetryOnError:     false,
```

分析上述代码，我们发现 **Reflector** 中的 **store** 其实是引用 **Controller.Config** 里的 **Queue** 属性，即 **fifo** 队列，而非 **ReplicationManager** 里的 **controllerStore**。我们费了这么大的劲，才弄明白这个简单的问题，这告诉我们一个事实：编程中有良好的命名规则很重要。

下面这段代码是 **Controller** 从队列 **Queue** 中拉取资源对象并且交给 **Controller.Config** 对象中的 **Process** 方法 **func (obj interface{})** 进行处理，从而最终完成了整个 **Controller** 框架的闭环过程。

```
func (c *Controller) processLoop() {  
    for {  
        obj := c.config.Queue.Pop()  
        err := c.config.Process(obj)  
        if err != nil {  
            if c.config.RetryOnError {  
                // This is the safe way to re-enqueue.  
                c.config.Queue.AddIfNotPresent(obj)  
            }  
        }  
    }  
}
```

至于上述过程的调用则是在Controller启动（Run方法）的最后一步里，Controller框架定时每秒调用一次上述函数，代码如下：

```
util.Until(c.processLoop, time.Second, stopCh)
```

最后，给读者留一个源码解读的问题，即ReplicationManager里除了RC Controller，又构造了一个用于Pod的Controller，它的逻辑具体是怎样实现的？它与RC Controller是怎样交互的？

6.3.3 设计总结

相对于之前的 Kubernetes API Server 设计来说，Kubernetes Controller Server 的设计没有那么复杂，而且精彩依旧。不愧是大师的作品，ControllerFramework 精巧细致的设计使得整个进程中各种资源对象的同步逻辑在代码实现方面保持了高度一致性与简捷性。此外，在关键资源 RC（ReplicationController）的同步逻辑中所采用的流控机制也简单、高效。

本节我们针对 Kubernetes Controller Server 中的精华部分——Controller Framework 的设计做一个整理分析。首先，framework.Controller 内部维护一个 Config 对象，保留了一个标准的消息、事件分发系统的三要素。

- 生产者：cache.ListerWatch。
- 队列：cache.cacheStore（Queue）。
- 消费者：用回调函数来模拟（framework.ResourceEventHandlerFuncs）。

由于生产者的逻辑比较复杂，在这个系统中也有其特殊性，即拉取资源并监控资源的变化，由此产生了真正的待处理任务，所以又设计了一个 ListerWatcher 接口，将底层的复杂逻辑“框架化”，放入 cache.Reflector 中，使用者只要简单地实现 ListerWatcher 接口的 ListFunc 与 WatchFunc 即可。另外，cache.Reflector 也是独立于 Controller Framework 的一个组件，隶属于 cache 包，它的功能是将任意资源对象

拉取到本地缓存中并监控资源的变化，保持本地缓存的同步，其目标是减轻对Kubernetes API Server的请求压力。

图6.4给出了ControllerFramework的整体架构设计图。

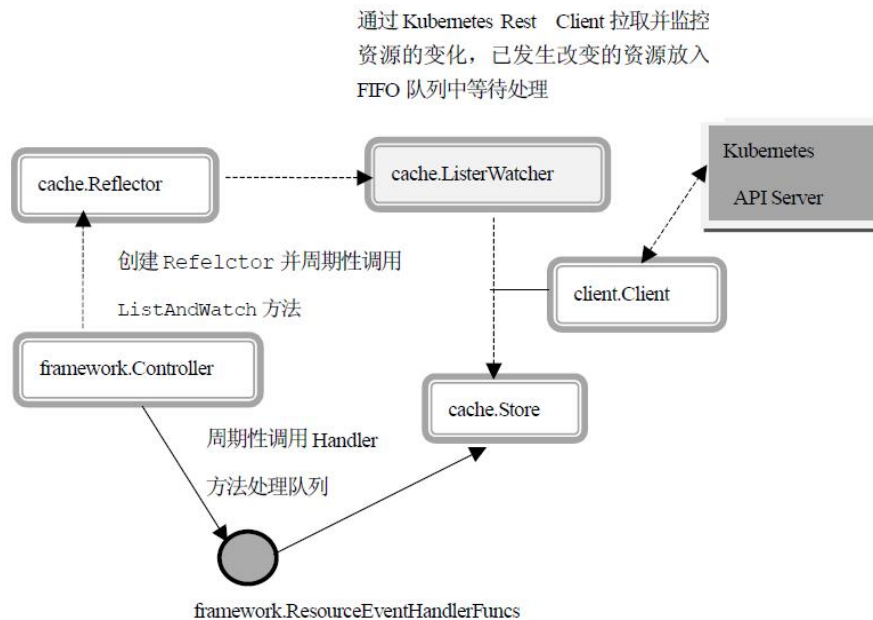


图6.4 Controller Framework的整体架构设计图

Kubernetes Controller Server 中所有涉及同步的资源都采用了Controller Framework框架来进行驱动，图6.5给出了整体设计示意图。

从图6.5可以看出，除了Node、Route、CloudService这三个资源依赖于Kubernetes所处的云计算环境，只能通过CloudProvider接口所提供的API来完成资源同步，其他资源都采用了Controller Framework框架来进行资源同步。图中的虚线箭头表示针对目标资源创建了一个framework.Controller对象，其中的某些资源如RC、PV、Tokens的同步过程需要获取并监听其他与之相关联的资源对象。这里只有ResourceQuota资源比较另类，它没有采用Controller Framework，一个

原因是 ResourceQuota 涉及很多资源对象，不大好应用 framework.Controller，另外一个原因可能是写 ResourceQuotaManager 的大牛拥有比较浪漫的情怀，看看下面这段 Kubernetes 中最优美的代码吧：

```
func (rm *ResourceQuotaManager) Run(period
time.Duration) {
    rm.syncTime = time.Tick(period)
    go util.Forever(func() { rm.synchronize() },
period)
}
```

核心代码翻译过来就是这个意思：从此他们过上了幸福的生活，一去不复返了！

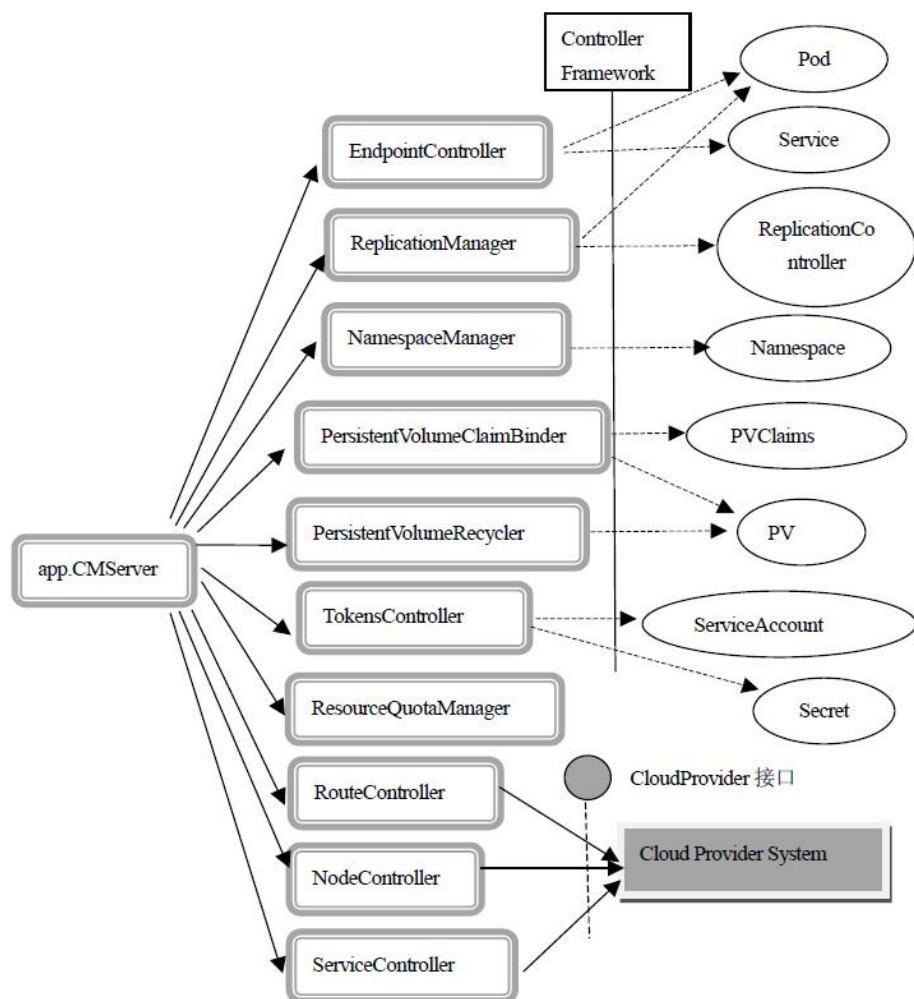


图6.5 Kubernetes Controller Server整体设计示意图

6.4 kube-scheduler进程源码分析

Kubernetes Scheduler Server是由kube-scheduler进程实现的，它运行在Kubernetes的管理节点——Master上并主要负责完成从Pod到Node的调度过程。Kubernetes Scheduler Server跟踪Kubernetes集群中所有Node的资源利用情况，并采取合适的调度策略，确保调度的均衡性，避免集群中的某些节点“过载”。从某种意义上来说，Kubernetes Scheduler Server也是Kubernetes集群的“大脑”。

谷歌作为公有云的重要供应商，积累了很多经验并且了解客户的需求。在谷歌看来，客户并不真正关心他们的服务究竟运行在哪台机器上，他们最关心服务的可靠性，希望发生故障后能自动恢复。遵循这一指导思想，Kubernetes Scheduler Server实现了“完全市场经济”的调度原则并彻底抛弃了传统意义上的“计划经济”。

下面我们分别对其启动过程、关键代码分析及设计总结等方面进行深入分析和讲解。

6.4.1 进程启动过程

kube-scheduler进程的入口类源码位置如下：

`github.com/GoogleCloudPlatform/kubernetes/plugin/cmd/kube-scheduler/scheduler.go`。

入口`main()`函数的逻辑如下：

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())
    s := app.NewSchedulerServer()
    s.AddFlags(pflag.CommandLine)
    util.InitFlags()
    util.InitLogs()
    defer util.FlushLogs()
    verflag.PrintAndExitIfRequested()
    s.Run(pflag.CommandLine.Args())
}
```

对上述代码的风格和逻辑我们再熟悉不过了：创建一个SchedulerServer对象，将命令行参数传入，并且进入SchedulerServer的Run方法，无限循环下去。

按照惯例，我们首先看看 SchedulerServer 的数据结构（app/server.go），下面是其定义：

```
type SchedulerServer struct {  
    Port          int  
    Address       util.IP  
    AlgorithmProvider string  
    PolicyConfigFile string  
    EnableProfiling bool  
    Master         string  
    Kubeconfig     string  
}
```

这里的关键属性有以下两个。

- **AlgorithmProvider**：对应参数 `algorithm-provider`，是 `AlgorithmProviderConfig` 的名称。
- **PolicyConfigFile**：用来加载调度策略文件。

从代码上来看这两个参数的作用其实是一样的，都是加载一组调度规则，这组调度规则要么在程序里定义为一个 `AlgorithmProviderConfig`，要么保存到文件中。下面的源码清楚地解释了这个过程：

```
func (s *SchedulerServer) createConfig(configFactory  
*factory.ConfigFactory) (*scheduler.Config, error) {  
    var policy schedulerapi.Policy  
    var configData []byte
```

```

        if _, err := os.Stat(s.PolicyConfigFile); err ==
nil {

                                configData, err =
ioutil.ReadFile(s.PolicyConfigFile)

                if err != nil {
                        return nil, fmt.Errorf("Unable to read
policy config: %v", err)
                }

                                err =
latestschedulerapi.Codec.DecodeInto(configData, &policy)

                if err != nil {
                        return nil, fmt.Errorf("Invalid
configuration: %v", err)
                }

                return configFactory.CreateFromConfig(policy)
        }

        // if the config file isn't provided, use the
specified (or default) provider

        // check of algorithm provider is registered and
fail fast

                                _, err :=
factory.GetAlgorithmProvider(s.AlgorithmProvider)

                if err != nil {
                        return nil, err

```

```

    }

    return
    configFactory.CreateFromProvider(s.AlgorithmProvider)
}

```

创建了 `SchedulerServer` 结构体实例后，调用此实例的方法 `func (s*APIServer) Run (_[]string)`，进入关键流程。首先，创建一个 `Rest Client`对象用于访问 `Kubernetes API Server`提供的 `API`服务：

```

    kubeClient, err := client.New(kubeconfig)
    if err != nil {
        glog.Fatalf("Invalid API configuration: %v",
err)
    }

```

随后，创建一个 `HTTP Server`以提供必要的性能分析（`Performance Profile`）和性能指标度量（`Metrics`）的 `Rest`服务：

```

go func() {
    mux := http.NewServeMux()
    healthz.InstallHandler(mux)
    if s.EnableProfiling {
        mux.HandleFunc("/debug/pprof/",
pprof.Index)
        mux.HandleFunc("/debug/pprof/profile",
pprof.Profile)
    }
}()

```

```

                                mux.HandleFunc("/debug/pprof/symbol",
pprof.Symbol)
                                }
                                mux.Handle("/metrics", prometheus.Handler())

                                server := &http.Server{
                                                                Addr:
net.JoinHostPort(s.Address.String(), strconv.Itoa(s.Port)),
                                Handler: mux,
                                }
                                glog.Fatal(server.ListenAndServe())
                                }()

```

接下来，启动程序构造了**ConfigFactory**，这个结构体包括了创建一个**Scheduler**所需的必要属性。

- **PodQueue**: 需要调度的Pod队列。
- **BindPodsRateLimiter**: 调度过程中限制Pod绑定速度的限速器。
- **modeler**: 这是用于优化Pod调度过程而设计的一个特殊对象，用于“预测未来”。一个Pod被计划调度到机器A的事实被称为**assumed**调度，即假定调度，这些调度安排被保存到特定队列里，此时调度过程是能看到这个预安排的，因而会影响到其他Pod的调度。
- **PodLister**: 负责拉取已经调度过的，以及被假定调度过的Pod列表。
- **NodeLister**: 负责拉取Node节点（Minion）列表。
- **ServiceLister**: 负责拉取Kubernetes服务列表。

- `ScheduledPodLister`、`scheduledPodPopulator`：Controller框架创建过程中返回的Store对象与controller对象，负责定期从Kubernetes API Server上拉取已经调度好的Pod列表，并将这些Pod从modeler的假定调度过的队列中删除。

在构造 `ConfigFactory` 的方法 `factory.NewConfigFactory(kubeClient)` 中，我们看到下面这段代码：

```
c.ScheduledPodLister.Store, c.scheduledPodPopulator =
framework.NewInformer(
    c.createAssignedPodLW(),
    &api.Pod{},
    0,
    framework.ResourceEventHandlerFuncs{
        AddFunc: func(obj interface{}) {
            if pod, ok := obj.(*api.Pod); ok {
                c.modeler.LockedAction(func() {
                    c.modeler.ForgetPod(pod)
                })
            }
        },
        DeleteFunc: func(obj interface{}) {
            c.modeler.LockedAction(func() {
                switch t := obj.(type) {
                case *api.Pod:
                    c.modeler.ForgetPod(t)
                case
```



```
cache.DeletedFinalStateUnknown:
```

```
c.modeler.ForgetPodByKey(t.Key)
    }
    })
},
},
)
```

这里沿用了之前看到的 **controller framework** 的身影，上述 **Controller** 实例所做的事情是获取并监听已经调度的 **Pod** 列表，并将这些 **Pod** 列表从 **modeler** 中的“**assumed**”队列中删除。

接下来，启动进程用上述创建好的 **ConfigFactory** 对象作为参数来调用 **SchdulerServer** 的 **createConfig** 方法，创建一个 **Scheduler.Config** 对象，而此段代码的关键逻辑则集中在 **ConfigFactory** 的 **CreateFromKeys** 这个函数里，其主要步骤如下。

(1) 创建一个与 **Pod** 相关的 **Reflector** 对象并定期执行，该 **Reflector** 负责查询并监测等待调度的 **Pod** 列表，即还没有分配主机的 **Pod** (**Unsigned Pod**)，然后把它们放入 **ConfigFactory** 的 **PodQueue** 中等待调度。相关代码为：`cache.NewReflector (f.createUnassignedPodLW (), &api.Pod{}, f.PodQueue, 0) .RunUntil (f.StopEverything)`。

(2) 启动 **ConfigFactory** 的 **scheduledPodPopulator Controller** 对象，负责定期从 **Kubernetes API Server** 上拉取已经调度好的 **Pod** 列表，并将这些 **Pod** 从 **modeler** 中的假定 (**assumed**) 调度过的队列中删除。相关代码为：`go f.scheduledPodPopulator.Run (f.StopEverything)`。

(3) 创建一个Node相关的Reflector对象并定期执行，该Reflector负责查询并监测可用的Node列表（可用意味着Node的spec.unschedulable属性为false），这些Node被放入ConfigFactory的NodeLister.Store里。相关代码为：`cache.NewReflector（ f.createMinionLW（ ） ， &api.Node{} ， f.NodeLister.Store ， 0）.RunUntil（f.StopEverything）`。

(4) 创建一个Service相关的Reflector对象并定期执行，该Reflector负责查询并监测已定义的Service列表，并放入ConfigFactory的ServiceLister.Store里。这个过程的目的目的是Scheduler需要知道一个Service当前所创建的所有Pod，以便能正确地进行调度。相关代码为：`cache.NewReflector（ f.createServiceLW（ ） ， &api.Service{} ， f.ServiceLister.Store， 0）.RunUntil（f.StopEverything）`。

(5) 创建一个实现了algorithm.ScheduleAlgorithm接口的对象genericScheduler，它负责完成从Pod到Node的具体调度工作，调度完成的Pod放入ConfigFactory的PodLister里。相关代码为：`algo :=scheduler.NewGenericScheduler（ predicateFuncs ， priorityConfigs ， f.PodLister， r）`。

(6) 最后一步，使用之前的这些信息创建Scheduler.Config对象并返回。

从上面的分析我们看出，其实在创建Scheduler.Config的过程中已经完成了Kubernetes Scheduler Server进程中的很多启动工作，于是整个进程的启动过程的最后一步简单明了：使用刚刚创建好的Config对象来构造一个Scheduler对象并启动运行。即下面的两行代码：

```
    sched := scheduler.New(config)
    sched.Run()
```

而Scheduler的Run方法就是不停地执行scheduleOne方法:

```
                                go    util.Until(s.scheduleOne,    0,
s.config.StopEverything)
```

scheduleOne方法的逻辑也比较清晰，即获取下一个待调度的Pod，然后交给genericScheduler进行调度（完成Pod到某个Node的绑定过程），调度成功以后通知Modeler。这个过程同时增加了限流和性能指标的逻辑。

6.4.2 关键代码分析

在6.4.1节对kube-scheduler进程的启动过程进行详细分析后，我们大致明白了Kubernetes Scheduler Server的工作流程，但由于代码中涉及多个Pod队列和Pod状态切换逻辑，因此这里有必要对这个问题进行详细分析，以弄清在整个调度过程中Pod的“来龙去脉”。首先，我们知道ConfigFactory里的PodQueue是“待调度的Pod队列”，这个过程是通过无限循环执行一个Reflector来从Kubernetes API Server上获取待调度的Pod列表并填充到队列中实现的，因为Reflector框架已经实现了通用的代码，所以到了Kubernetes Scheduler Server这里，通过一行代码就能完成这个复杂的过程：

```
cache.NewReflector(f.createUnassignedPodLW(),
&api.Pod{}, f.PodQueue, 0).RunUntil(f.StopEverything)
```

上述代码中的createUnassignedPodLW是查询和监测spec.nodeName为空的Pod列表，此外，我们注意到scheduler.Config里提供了NextPod这个函数指针来从上述队列中消费一个元素，下面是相关代码片段（来自ConfigFactory的CreateFromKeys方法中创建scheduler.Config的代码）：

```
NextPod: func() *api.Pod {
    pod := f.PodQueue.Pop().(*api.Pod)
    glog.V(2).Infof("About to try and schedule
pod %v", pod.Name)
```

```
        return pod
    },
```

然后，这个PodQueue是怎样被消费的呢？就在之前提到的Scheduler.scheduleOne的方法里，每次调用NextPod方法会获取一个可用的Pod，然后交给genericScheduler进行调度，下面是相关代码片段（省略了其他代码）：

```
    pod := s.config.NextPod()
    if s.config.BindPodsRateLimiter != nil {
        s.config.BindPodsRateLimiter.Accept()
    }
    dest, err := s.config.Algorithm.Schedule(pod,
s.config.MinionLister)
```

genericScheduler.Schedule方法只是给出该Pod调度到的目标Node，如果调度成功，则设置该Pod的spec.nodeName为目标Node，然后通过HTTP Rest调用写入Kubernetes API Server里完成Pod的Binding操作，最后通知ConfigFactory的modeler（具体实例对应scheduler.SimpleModeler），将此Pod放入Assumed Pod队列，下面是相关代码片段：

```
    s.config.Modeler.LockedAction(func() {
        bindingStart := time.Now()
        err := s.config.Binder.Bind(b)

        metrics.BindingLatency.Observe(metrics.SinceInMicroseconds(
```

```

bindingStart))

        s.config.Recorder.Eventf(pod, "scheduled",
        "Successfully assigned %v to %v", pod.Name, dest)

        // tell the model to assume that this
        binding took effect.

        assumed := *pod
        assumed.Spec.NodeName = dest
        s.config.Modeler.AssumePod(&assumed)

    })

```

当Pod执行Bind操作成功以后，Kubernetes API Server上Pod已经满足“已调度”的条件，因为spec.nodeName已经被设置为目标Node地址，此时ConfigFactory的scheduledPodPopulator这个Controller就会监听到此变化，将此Pod从modeler中的Assumed队列中删除，下面是相关代码片段：

```

framework.ResourceEventHandlerFuncs{
    AddFunc: func(obj interface{}) {
        if pod, ok := obj.
(*api.Pod); ok {

        c.modeler.LockedAction(func() {

        c.modeler.ForgetPod(pod)

        })

    }
}

```

```
    },  
    .....  
},
```

谷歌的大神在源码中说明Modeler的存在是为了调度的优化，那么这个优化具体体现在哪里呢？由于Rest Watch API存在延时，当前已经调度好的Pod很可能还未被通知给Scheduler，于是大神灵光一闪：为每个刚刚调度完成的Pod发放一个“暂住证”，安排“暂住”到“Assumed”队列里，然后设计一个获取当前“已调度”的Pod队列的新方法，该方法合并Assumed队列与Watch缓存队列，这样一来，就得到了最佳答案。如果你打算看看这段代码，那么它就在SimpleModeler的listPods方法里，至此，你若也完全明白了c.PodLister=modeler.PodLister()这句简单却又深奥的代码，那么恭喜你，你离大神的距离又缩短了一个厘米。

接下来，我们深入分析Pod调度中所用到的流控技术，缘起于下面这段代码：

```
if s.config.BindPodsRateLimiter != nil {  
    s.config.BindPodsRateLimiter.Accept()  
}
```

上述代码中的BindPodsRateLimiter采用了开源项目juju的一个子项目ratelimit，项目地址为<https://github.com/juju/ratelimit>，它实现了一个高效的基于经典令牌桶（Token Bucket）的流控算法。如图6.6所示是经典令牌桶流控算法的原理示意图。

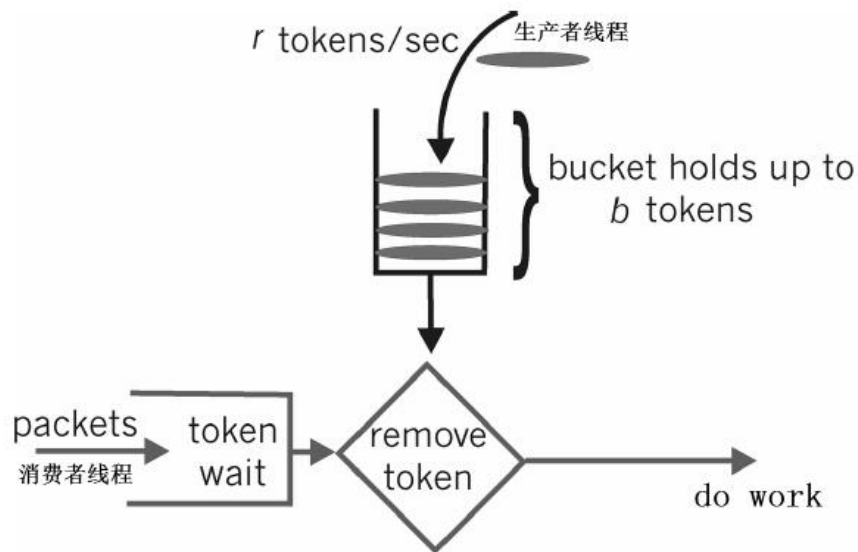


图6.6 令牌桶流控算法示意图

简单地说，控制线程以固定速率向一个固定容量的桶（**Bucket**）中投放令牌（**Token**），消费者线程则等待并获取到一个令牌后才能继续接下来的任务，否则需要等待可用令牌的到来。具体说来，假如用户配置的平均限流速率为 r ，则每隔 $1/r$ 秒就会有一个令牌被加入桶中，而令牌桶最多可以存储 b 个令牌，如果令牌到达时令牌桶已经满了，那么这个令牌会被丢弃。从长期运行结果来看，消费者的处理速率被限制成常量 r 。令牌桶流控算法除了能够限制平均处理速度，还允许某种程度的突发速率。

juju的ratelimit模块通过下面的API提供了构造一个令牌桶的简单做法，其中，`rate`参数表示每秒填充到桶里的令牌数量，`capacity`则是桶的容量：

```
func NewBucketWithRate(rate float64, capacity int64)
*Bucket
```

我们回头再看看Kubernetes SchedulerServer中BindPodsRateLimiter的赋值代码：`c.BindPodsRateLimiter=util.NewTokenBucketRateLimiter (BindPodsQps, BindPodsBurst)`，跟踪进去，发现它就是调用了刚才所提到的juju函数`limiter := ratelimit.NewBucketWithRate (float64 (qps), int64 (burst))`，其中qps目前为常量15，而burst为20，目前在Kubernetes 1.0版本中还没有提供命令行参数来配置此变量，会在未来的版本中实现。

最后，我们一起深入分析Kubernetes Scheduler Server中关于Pod调度的细节。首先，我们需要理解启动过程中SchedulerServer加载调度策略相关配置的这段代码：

```
                                predicateFuncs,      err      :=
getFitPredicateFunctions(predicateKeys, pluginArgs)
                                priorityConfigs,    err      :=
getPriorityFunctionConfigs(priorityKeys, pluginArgs)
    algo := scheduler.NewGenericScheduler(predicateFuncs,
priorityConfigs, f.PodLister, r)
```

这里加载了两组策略，其中predicateFuncs是一个Map，key为FitPredicate的名称，value为对应的algorithm.FitPredicate函数，它表明一个候选的Node是否满足当前Pod的调度要求，FitPredicate函数的具体定义如下：

```
    type FitPredicate func(pod *api.Pod, existingPods
[]*api.Pod, node string) (bool, error)
```

FitPredicate是Pod调度过程中必须满足的规则，只有顺利通过由所有**FitPredicate**组成的这道封锁线，一个**Node**才能拿到主会场的“入场券”，成为一个合格的“候选人”，等待下一步“评审”。目前系统提供的具体的**FitPredicate**实现都在 **predicates.go** 里，系统默认加载注册**FitPredicate**的地方在**defaultPredicates**方法里。

当有一组**Node**通过筛查成为“候选人”之后，需要有一种办法来选择“最优”的**Node**，这就是接下来我们要介绍的**priorityConfigs**所要做的事情了。**priorityConfigs**是一个数组，类型为**algorithm.PriorityConfig**，**PriorityConfig**包括一个**PriorityFunction**函数，用来计算并给出一组**Node**的优先级，下面是相关代码：

```
type PriorityConfig struct {
    Function PriorityFunction
    Weight    int
}

type PriorityFunction func(pod *api.Pod, podLister
PodLister, minionLister MinionLister) (HostPriorityList,
error)

type HostPriorityList []HostPriority
func (h HostPriorityList) Len() int {
    return len(h)
}

func (h HostPriorityList) Less(i, j int) bool {
    if h[i].Score == h[j].Score {
        return h[i].Host < h[j].Host
    }
}
```

```
        return h[i].Score < h[j].Score
    }
}
```

如果看到这里还是不太明白它的用途，那么认真读一读下面这段来自`genericScheduler`的计算候选节点优先级的`PrioritizeNodes`方法，你就能顿悟了：一个候选节点的优先级总分是所有评委老师（`PriorityConfig`）一起给出的“加权总分”，评委老师越是德高望重（`PriorityConfig.Weight`越大），他的评分影响力就越大：

```
combinedScores := map[string]int{}
for _, priorityConfig := range priorityConfigs {
    weight := priorityConfig.Weight
    // skip the priority function if the weight is
specified as 0
    if weight == 0 {
        continue
    }
    priorityFunc := priorityConfig.Function
    prioritizedList, err := priorityFunc(pod,
podLister, minionLister)
    if err != nil {
        return algorithm.HostPriorityList{}, err
    }
    for _, hostEntry := range prioritizedList {
        combinedScores[hostEntry.Host] +=
hostEntry.Score * weight
    }
}
```

```
    }
    for host, score := range combinedScores {
        glog.V(10).Infof("Host %s Score %d", host,
score)

                                result = append(result,
algorithm.HostPriority{Host: host, Score: score})
    }
    return result, nil
```

接下来，我们看看系统初始化加载的默认的Predicate与Priorities有哪些，通过追踪代码，我们发现默认加载的代码位于plugin/pkg/scheduler/algorithmprovider/default/default.go的init函数里：

```
func init() {

factory.RegisterAlgorithmProvider(factory.DefaultProvider,
defaultPredicates(), defaultPriorities())

    // EqualPriority is a prioritizer function that
gives an equal weight of one to all minions

    // Register the priority function so that its
available

    // but do not include it as part of the default
priorities

    factory.RegisterPriorityFunction("EqualPriority",
scheduler.EqualPriority, 1)

}
```

跟踪进去后，我们看到系统默认加载的predicates有如下几种：

- PodFitsResources;
- MatchNodeSelector;
- HostName °

而默认加载的priorities则有如下几种：

- LeastRequestedPriority;
- BalancedResourceAllocation;
- ServiceSpreadingPriority °

从上述这些信息来看，Kubernetes默认的调度指导原则是尽量均匀分布Node到不同的Node上，并且确保各个Node上的资源利用率基本保持一致，也就是说如果你有100台机器，则可能每个机器都被调度到，而不是只有其中的20%被调度到，哪怕每台机器都只利用了不到10%的资源，这不正是所谓的“韩信点兵，多多益善”么？

接下来我们以服务亲和性这个默认没有加载的Predicate为例，看看Kubernetes是如何通过Policy文件注册加载它的。下面是我们定义的一个Policy文件：

```
{
  "kind" : "Policy",
  "version" : "v1",
  "predicates" : [
    .....
    {"name" : "RegionZoneAffinity", "argument"
: {"serviceAffinity" : {"labels" : ["region", "zone"]}}}
]
```

```

    ],
    "priorities" : [
        .....
        {"name" : "RackSpread", "weight" : 1,
"argument" : {"serviceAnti
Affinity" : {"label" : "rack"}}}
    ]
}

```

首先，这个文件被映射成 `api.Policy` 对象（`plugin/pkg/scheduler/api/types.go`）。下面是其结构体定义：

```

type Policy struct {
    api.TypeMeta `json:",inline"`
    // Holds the information to configure the fit
predicate functions
    Predicates []PredicatePolicy `json:"predicates"`
    // Holds the information to configure the priority
functions
    Priorities []PriorityPolicy `json:"priorities"`
}

```

我们看到 `policy` 文件中的 `predicates` 部分被映射为 `PredicatePolicy` 数组：

```

type PredicatePolicy struct {
    Name string `json:"name"`
}

```

```
    Argument *PredicateArgument 'json:"argument"'
}
```

而PredicateArgument的定义如下，包括服务亲和性的相关属性ServiceAffinity:

```
type PredicateArgument struct {
    ServiceAffinity *ServiceAffinity
    `json:"serviceAffinity"`
    LabelsPresence *LabelsPresence
    `json:"labelsPresence"`
}
```

策略文件被映射为api.Policy对象后，PredicatePolicy部分的处理逻辑则交给下面的函数进行处理(plugin/pkg/scheduler/factory/plugin.go)：

```
func RegisterCustomFitPredicate(policy
schedulerapi.PredicatePolicy) string {
    var predicateFactory FitPredicateFactory
    var ok bool
    validatePredicateOrDie(policy)
    // generate the predicate function, if a custom
type is requested
    if policy.Argument != nil {
        if policy.Argument.ServiceAffinity != nil {
            predicateFactory = func(args
```

```

PluginFactoryArgs) algorithm.FitPredicate {
                                                    return
predicates.NewServiceAffinityPredicate(
                                                    args.PodLister,
                                                    args.ServiceLister,
                                                    args.NodeInfo,

policy.Argument.ServiceAffinity.Labels,
                                                    )
}
} else if policy.Argument.LabelsPresence !=
nil {
                                                    predicateFactory = func(args
PluginFactoryArgs) algorithm.FitPredicate {
                                                    return
predicates.NewNodeLabelPredicate(
                                                    args.NodeInfo,

policy.Argument.LabelsPresence.Labels,

policy.Argument.LabelsPresence.Presence,
                                                    )
}
}
}

```

在上面的代码中，当 **ServiceAffinity** 属性不空时，就会调用 **predicates.NewServiceAffinityPredicate** 方法来创建一个处理服务亲和性

的FitPredicate，随后被加载到全局的predicateFactory中生效。

最后，genericScheduler.Schedule方法才是真正实现Pod调度的方法，我们看看这段完整代码：

```
func (g *genericScheduler) Schedule(pod *api.Pod,
minionLister algorithm.MinionLister) (string, error) {
    minions, err := minionLister.List()
    if err != nil {
        return "", err
    }
    if len(minions.Items) == 0 {
        return "", ErrNoNodesAvailable
    }

    filteredNodes, failedPredicateMap, err :=
findNodesThatFit(pod, g.pods, g.predicates, minions)
    if err != nil {
        return "", err
    }

    priorityList, err := PrioritizeNodes(pod, g.pods,
g.prioritizers, algorithm.FakeMinionLister(filteredNodes))
    if err != nil {
        return "", err
    }
    if len(priorityList) == 0 {
```

```
        return "", &FitError{
            Pod: pod,
            FailedPredicates: failedPredicateMap,
        }
    }

    return g.selectHost(priorityList)
}
```

这段代码已经简单得不能再简单了，因为该干的活都已经被 **predicates** 与 **priorities** 干完了！架构之美，就在于程序逻辑分解得恰到好处，每个组件各司其职，从而化繁为简，使得主体流程清晰直观，犹如行云流水，一气呵成。

向谷歌大神们致敬！

6.4.3 设计总结

与之前的Kubernetes API Server和Kubernetes Controller Manager对比，Kubernetes Scheduler Server的设计和代码显得更为“精妙”。项目中引入ratelimit组件来解决Pod调度的流控问题的做法，既大大简化了代码量，又体现了大神们的气度。

Kubernetes Scheduler Server的一个关键设计目标是“插件化”，以方便Cloud Provider或者个人用户根据自己的需求进行定制，本节我们围绕其中最为关键的“FitPredicate与PriorityFunction”对其设计做一个总结。如图6.7所示，在plugin.go中采用了全局变量的Map变量记录了系统当前注册的FitPredicate与PriorityFunction，其中fitPredicateMap和priorityFunctionMap分别存放FitPredicateFactory与PriorityConfigFactory（包含了PriorityFunctionFactory的一个引用）中。可以看出，这里的设计采用了标准的工厂模式，factory.PluginFactoryArgs这个数据结构可以认为是一个上下文环境变量，它提供给PluginFactory必要的数据访问接口，比如获取一个Node的详细信息并获取一个Pod上的所有Service信息等，这些接口可以被某些具体的FitPredicate或PriorityFunction使用，以实现特定的功能，如图6.7所示的predicates.PodFitsPods和priorities.LeastRequestedPriority就分别使用了上述接口。

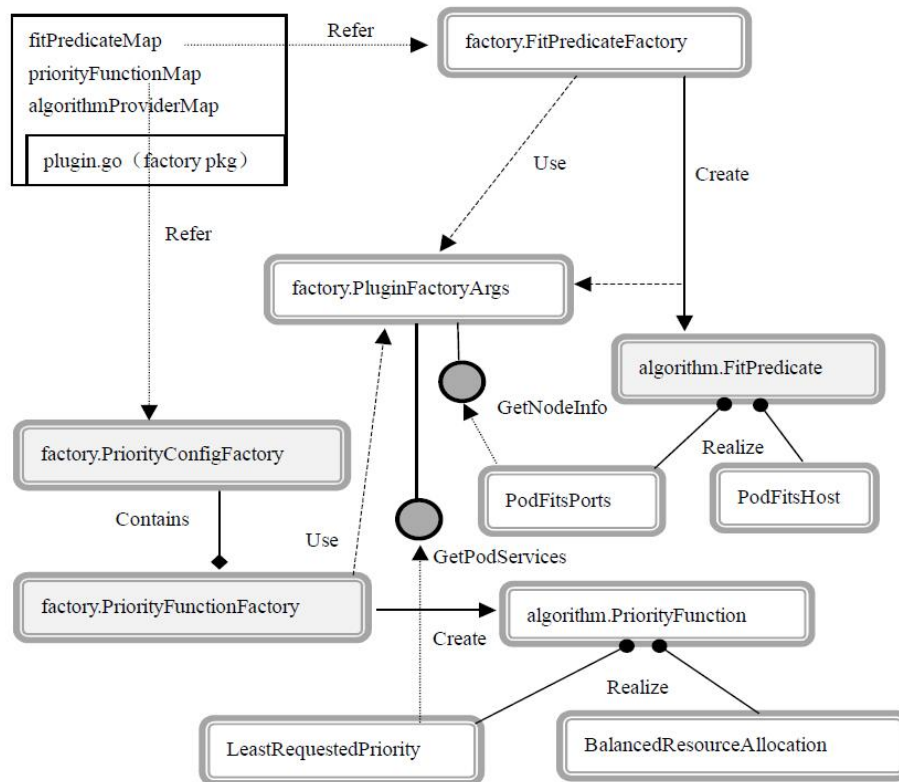


图6.7 Kubernetes Scheduler Server调度策略相关设计示意图

我们注意到PluginFactoryArgs的接口都是Kubernetes的资源访问接口，那么问题就来了，为何不直接用Kubernetes RestClient API访问呢？一个主要的原因是如果这样做，则增加了插件开发者开发和调测的难度，因为开发者需要再去学习和掌握RestClient；另外一个原因是效率的问题，如果大家都采用框架提供的“标准方法”查询资源，那么框架可以实现很多优化，比较容易缓存；最后一个原因则与之前我们分析的“Assumed Pod”有关，即查询当前已经调度过的Pod列表是有其特殊性的，PluginFactoryArgs中的PodLister方法就是引用了ConfigFactory的PodLister。

algorithmProviderMap这个全局变量则保存了一组命名的调度策略配置文件（AlgorithmProviderConfig），其实就是一组FitPredicate与PriorityFunction的集合，其定义如下：

```
type AlgorithmProviderConfig struct {  
    FitPredicateKeys    util.StringSet  
    PriorityFunctionKeys util.StringSet  
}
```

它的作用是预配置和自定义调度规则，Kubernetes Scheduler Server默认加载了一个名为“DefaultProvider”的调度策略配置，通过定义和加载不同的调度规则配置文件，我们可以改变默认的调度策略，比如我们可以定义两组规则文件：其中一个命名为“function_test_cfg”，面向功能测试，调度原则是尽量在最少的机器上调度Pod以节省资源；另外一个则命名为performance_test_cfg”，面向性能测试，调度原则是尽可能使用更多的机器，以测试系统性能。

顺便提一下，笔者认为在Kubernetes Scheduler Server中关于PredicateArgument/PriorityArgument的设计并不好，这里没有将Predicate的属性通用化，比如采用key-value这种模式，因此导致Policy文件格式与Predicate/Priority关联之间的强耦合性，增加了代码理解的困难性，之前分析的Policy文件中服务亲和性的Predicate的加载逻辑即反映了这个问题，笔者深信，未来版本中大神们会认真考虑重构问题。

至此，Master节点上的进程的源码都已经分析完毕，我们发现这些进程所做的事情，归根到底就是两件事：Pod调度+智能纠错，这也是为什么这些进程所在的节点被称为“Master”，因为它们高高在上，

运筹帷幄。虽然“**Master**”从不深入底层微服私访，但也的确鞠躬尽瘁、日理万机，计算机的世界果然比我们人类的世界要单纯、高效很多，真心希望人工智能的发展不会让它们的世界也变得扑朔迷离。

6.5 kubelet进程源码分析

kubelet是运行在**Minion**节点上的重要守护进程，是工作在一线的重要“工人”，它才是负责“实例化”和“启动”一个具体的**Pod**的幕后主导，并且掌管着本节点上的**Pod**和容器的全生命周期过程，定时向**Master**汇报工作情况。此外，**kubelet**进程也是一个“**Server**”进程，它默认监听10250端口，接收并执行远程（**Master**）发来的指令。

下面我们分别对其启动过程、关键代码分析及设计总结等方面进行深入分析和讲解。

6.5.1 进程启动过程

kubelet进程的入口类源码位置如下:

github.com/GoogleCloudPlatform/kubernetes/cmd/kubelet/kubelet.go

入口main () 函数的逻辑如下:

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())
    s := app.NewKubeletServer()
    s.AddFlags(pflag.CommandLine)
    util.InitFlags()
    util.InitLogs()
    defer util.FlushLogs()
    verflag.PrintAndExitIfRequested()
    if err := s.Run(pflag.CommandLine.Args()); err !=
nil {
        fmt.Fprintf(os.Stderr, "%v\n", err)
        os.Exit(1)
    }
}
```

我们已经是第4次“遇见”这样的代码风格了，代码的颜值匹配度高达99%，这至少说明一点：谷歌在源码一致性方面做得很好，N多人写的代码看起来就好像出自一个人之手。我们先来看看KubeletServer这个结构体所包括的属性吧，这些属性可以分为以下几组。

1) 基本配置

- KubeConfig: kubelet默认配置文件路径。
- Address 、 Port 、 ReadOnlyPort 、 CadvisorPort 、 HealthzPort 、 HealthzBindAddress: 为kubelet绑定监听的地址，包括自身Server的地址，Cadvisor绑定的地址，以及自身健康检查服务的绑定地址等。
- RootDirectory 、 CertDirectory : kubelet 默认的工作目录 (/var/lib/kubelet)，用于存放配置及VM卷等数据，CertDirectory用于存放证书目录。

2) 管理Pod和容器相关的参数

- PodInfraContainerImage: Pod的infra容器的镜像名称，谷歌被屏蔽的时候可以换成自己的私有仓库的镜像名。
- CgroupRoot: 可选项，创建Pod的时候所使用的顶层的cgroup名字 (Root Cgroup) 。
- ContainerRuntime 、 DockerDaemonContainer 、 SystemContainer : 这三个参数分别表示选择什么容器技术 (Docker或者RKT)、Docker Daemon容器的名字及可选的系统资源容器名称，用来将所有非kernel的、不在容器中的进程放入此容器中。

3) 同步和自动运维相关的参数

- SyncFrequency、FileCheckFrequency、HTTPCheckFrequency: Pod容器同步周期、当前运行的容器实例分别与Kubernetes注册表中的信息、本地的Pod定义文件及以HTTP方式提供信息的数据源进行对比同步。
- RegistryPullQPS、RegistryBurst: 从注册表拉取待创建的Pod列表时的流控参数。
- NodeStatusUpdateFrequency: kubelet多久汇报一次当前Node的状态。
- ImageGCHighThresholdPercent、ImageGCLowThresholdPercent、LowDiskSpaceThresholdMB: 分别是Image镜像占用磁盘空间的高低水位阈值及本机磁盘最小空闲容量，当可用容量低于这个容量时，所有新Pod的创建请求会被拒绝。
- MaxContainerCount、MaxPerPodContainerCount: 分别是maximum-dead-containers与maximum-dead-containers-per-container，表示保留多少个死亡容器的实例在磁盘上，因为每个实例都会占用一定的磁盘，所以需要控制，默认是MaxContainerCount为100，MaxPerPodContainerCount为2，即每个容器保留最多两个死亡实例，每个Node保留最多100个死亡实例。

只要分析一下上述KubeletServer结构体的关键属性，我们就可以得到这样一个推论：kubelet进程的“工作量”还是很饱满的，一点都不比Master上的API Server、Controll Manager、Scheduler做得少。

在继续下面的代码分析之前，我们先要理解这里的一个重要概念“Pod Source”，它是kubelet用于获取Pod定义和描述信息的一个“数据

源”，kubelet进程查询并监听Pod Source来获取属于自己所在节点的Pod列表，当前支持三种Pod Source类型。

- Config File: 本地配置文件作为Pod数据源。
- Http URL: Pod数据源的内容通过一个HTTP URL方式获取。
- Kubernetes API Server: 默认方式，从API Server获取Pod数据源。

进程根据启动参数创建了KubeletServer以后，调用KubeletServer的run方法，进入启动流程，在流程的一开始首先设置了自身进程的oom_adj参数（默认为-900），这是利用了Linux的OOM机制，当系统发生OOM时，oom_adj的值越小，越不容易被系统Kill掉。

```
        if err := util.ApplyOomScoreAdj(0, s.OOMScoreAdj); err
!= nil {
            glog.Warning(err)
        }
```

为什么在之前的Master节点进程上都没有见到这个调用，而在kubelet进程上却看到这段逻辑？答案很简单，因为Master节点不运行Pod和容器，主机资源通常是稳定和宽裕的，而Minion节点由于需要运行大量的Pod和容器，因此容易产生OOM问题，所以这里要确保“守护者”不会因此而被系统Kill掉。

由于kubelet会跟API Server打交道，所以接下来创建了一个Rest Client对象来访问API Server。随后，启动进程构造了cAdvisor来监控本地的Docker容器，cAdvisor具体的创建代码则位于pkg/kubelet/cadvisor/cadvisor_linux.go里，引用了github.com/google/cadvisor这个同样属于谷歌开源的项目。

接着，初始化CloudProvider，这是因为如果Kubernetes运行在某个运营商的Cloud环境中，则很多环境和资源都需要从CloudProvider中获取，比如在创建Pod的过程中可能需要知道某个Node的真实主机名。

虽然容器可以绑定宿主机的网络空间，但若不当使用会导致系统安全漏洞，所以KubeletServer中的HostNetworkSources的属性用来控制哪些Pod允许绑定宿主机的网络空间，默认是都禁止绑定。举例说明，比如设置HostNetworkSources=api，http，则表明当一个Pod的定义源来自Kubernetes API Server或者某个HTTP URL时，则允许此Pod绑定到宿主机的网络空间。下面这行代码即上述处理逻辑中的一小部分：

```
                                hostNetworkSources,      err      :=  
kubelet.GetValidatedSources(strings.Split(s.HostNetworkSources, ","))
```

接下来加载数字证书，如果没有提供证书和私钥，则默认创建一个自签名的X509证书并保存到本地。下一步，创建一个Mounter对象，用来实现容器的文件系统挂载功能。

接下来的这段代码根据指定了DockerExecHandlerName参数的值，确定dockerExecHandler是采用Docker的exec命令还是nsenter来实现，默认采用了Docker的exec这种本地方式，Docker从1.3版本开始提供了exec指令，为进入容器内部提供了更好的手段。

```
var dockerExecHandler dockertools.ExecHandler  
switch s.DockerExecHandlerName {  
case "native":
```

```

                                dockerExecHandler =
&dockertools.NativeExecHandler{}
                                case "nsenter":
                                dockerExecHandler =
&dockertools.NsenterExecHandler{}
                                default:
                                log.Warningf("Unknown Docker exec handler
%q; defaulting to native", s.DockerExecHandlerName)
                                dockerExecHandler =
&dockertools.NativeExecHandler{}
                                }

```

运行至此，程序构造了一个KubeletConfig结构体，90%的变量与之前的KubeletServer一样，这让代码长度增加了20多行！定睛一看，源码上有TODO注释：“它应该可能被合并到KubeletServer里.....”，目测注释是另外一个大神添加的，这让笔者陷入了深深的思考：难道谷歌的绩效考评系统中也有恶俗的代码行数考核指标？

KubeletConfig创建好以后作为参数调用RunKubelet (&kcfg, nil)方法，程序运行到这里，才真正进入流程的核心步骤。下面这段代码表明kubelet会把自己的事件通知API Server:

```

eventBroadcaster := record.NewBroadcaster()
                                kcfg.Recorder =
eventBroadcaster.NewRecorder(api.EventSource{Component:
"kubelet", Host: kcfg.NodeName})
eventBroadcaster.StartLogging(glog.V(3).Infof)

```

```

        if kcfg.KubeClient != nil {
            glog.V(4).Infof("Sending events to api
server.")

eventBroadcaster.StartRecordingToSink(kcfg.KubeClient.Event
s(""))

        } else {
            glog.Warning("No api server defined - no
events will be sent to API server.")
        }

```

接下来，启动进程进入关键函数`createAndInitKubelet`中，这里首先创建一个`PodConfig`对象，并根据启动参数中`Pod Source`参数是否提供，来创建相应类型的`Pod Source`对象，这些`PodSource`在各种协程中运行，拉取`Pod`信息并汇总输出到同一个`Pod Channel`中等待`kubelet`处理。创建`PodConfig`的具体代码如下：

```

func makePodSourceConfig(kc *KubeletConfig)
*config.PodConfig {
    // source of all configuration

                                cfg :=
config.NewPodConfig(config.PodConfigNotificationSnapshotAnd
Updates, kc.Recorder)

    // define file config source
    if kc.ConfigFile != "" {
        glog.Infof("Adding manifest file: %v",

```

```

kc.ConfigFile)
                                config.NewSourceFile(kc.ConfigFile,
kc.NodeName,                                kc.FileCheckFrequency,
cfg.Channel(kubelet.FileSource))
    }

    // define url config source
    if kc.ManifestURL != "" {
        glog.Infof("Adding manifest url: %v",
kc.ManifestURL)
                                config.NewSourceURL(kc.ManifestURL,
kc.NodeName,                                kc.HTTPCheckFrequency,
cfg.Channel(kubelet.HTTPSource))
    }
    if kc.KubeClient != nil {
        glog.Infof("Watching apiserver")
        config.NewSourceApiserver(kc.KubeClient,
kc.NodeName, cfg.Channel(kubelet.ApiserverSource))
    }
    return cfg
}

```

然后，创建一个kubernetes并宣告它的诞生：

```

k, err = kubelet.NewMainKubelet(...)
k.BirthCry()

```

接着，触发kubelelet开启垃圾回收协程以清理无用的容器和镜像，释放磁盘空间，下面是其代码片段：

```
// Starts garbage collection threads.
func (kl *Kubelet) StartGarbageCollection() {
    go util.Forever(func() {
        if err := kl.containerGC.GarbageCollect(); err
        != nil {
            glog.Errorf("Container garbage collection
failed: %v", err)
        }
    }, time.Minute)

    go util.Forever(func() {
        if err := kl.imageManager.GarbageCollect();
        err != nil {
            glog.Errorf("Image garbage collection
failed: %v", err)
        }
    }, 5*time.Minute)
}
```

createAndInitKubelet方法创建kubelelet实例以后，返回到RunKubelet方法里，接下来调用startKubelet方法，此方法首先启动一个协程，让kubelelet处理来自PodSource的Pod Update消息，然后启动Kubelet Server，下面是具体代码：

```
func startKubelet(k KubeletBootstrap, podCfg
*config.PodConfig, kc *KubeletConfig) {
    // start the kubelet
    go util.Forever(func() { k.Run(podCfg.Updates())
}, 0)

    // start the kubelet server
    if kc.EnableServer {
        go util.Forever(func() {
            k.ListenAndServe(net.IP(kc.Address),
kc.Port, kc.TLSOptions, kc.EnableDebuggingHandlers)
        }, 0)
    }
    if kc.ReadOnlyPort > 0 {
        go util.Forever(func() {

k.ListenAndServeReadOnly(net.IP(kc.Address),
kc.ReadOnlyPort)
        }, 0)
    }
}
```

至此，kubelet进程启动完毕。

6.5.2 关键代码分析

6.5.1 节里，我们分析了 kubelet 进程的启动流程，大致明白了 kubelet 的核心工作流程就是不断从 PodSource 中获取与本节点相关的 Pod，然后开始“加工处理”，所以，我们先来分析 Pod Source 部分的代码。前面我们提到，kubelet 可以同时支持三类 Pod Source，为了能够将不同的 Pod Source“汇聚”到一起统一处理，谷歌特地设计了 PodConfig 这个对象，其代码如下：

```
type PodConfig struct {  
    pods *podStorage  
    mux  *config.Mux  
  
    // the channel of denormalized changes passed to  
listeners  
    updates chan kubelet.PodUpdate  
  
    // contains the list of all configured sources  
    sourcesLock sync.Mutex  
    sources     util.StringSet  
}
```

其中，sources 属性包括了当前加载的所有 Pod Source 类型，sourcesLock 是 source 的排他锁，在新增 Pod Source 的方法里使用它来避

免共享冲突。

当Pod发生变动时，例如Pod创建、删除或者更新，相关的Pod Source就会产生对应的PodUpdate事件并推送到Channel上。为了能够统一处理来自多个Source的Channel，谷歌设计了config.Mux这个“聚合器”，它负责监听多路Channel，当接收到Channel发来的事件以后，交给Merger对象进行统一处理，Merger对象最终把多路Channel发来的事件合并写入updates这个汇聚Channel里等待处理。

下面是config.Mux的结构体定义，其属性sources为一个Channel Map，key是对应的Pod Source的类型：

```
type Mux struct {  
    // Invoked when an update is sent to a source.  
    merger Merger  
    // Sources and their lock.  
    sourceLock sync.RWMutex  
    // Maps source names to channels  
    sources map[string]chan interface{}  
}
```

我们继续深入分析config.Mux的工作过程，前面提到，kubelet在启动过程中在makePodSourceConfig方法里创建了一个PodConfig对象，并且根据启动参数来决定要加载哪些类型的Pod Source，在这个过程中调用了下述方法来创建一个对应的Channel：

```
func (c *PodConfig) Channel(source string) chan<-  
interface{} {
```

```
    c.sourcesLock.Lock()
    defer c.sourcesLock.Unlock()
    c.sources.Insert(source)
    return c.mux.Channel(source)
}
```

而Channel具体的创建过程则在config.Mux里，Channel创建完成后被加入config.Mux的sources里并且启动一个协程开始监听消息，代码如下：

```
func (m *Mux) Channel(source string) chan interface{}{
{
    if len(source) == 0 {
        panic("Channel given an empty name")
    }
    m.sourceLock.Lock()
    defer m.sourceLock.Unlock()
    channel, exists := m.sources[source]
    if exists {
        return channel
    }
    newChannel := make(chan interface{})
    m.sources[source] = newChannel
    go util.Forever(func() { m.listen(source,
newChannel) }, 0)
    return newChannel
}
```

`config.Mux`的上述`listen`方法很简单，就是监听新创建的`Channel`，一旦发现`Channel`上有数据就交给`Merger`进行处理：

```
func (m *Mux) listen(source string, listenChannel <-
chan interface{}) {
    for update := range listenChannel {
        m.merger.Merge(source, update)
    }
}
```

我们先来看看`Pod Source`是如何发送`PodUpdate`事件到自己所在的`Channel`上的，在6.5.1节中我们所见到的下面这段代码创建了一个`Config File`类型的`Pod Source`：

```
// define file config source
if kc.ConfigFile != "" {
    glog.Infof("Adding manifest file: %v",
kc.ConfigFile)
    config.NewSourceFile(kc.ConfigFile,
kc.NodeName, kc.FileCheckFrequency,
cfg.Channel(kubelet.FileSource))
}
```

在`NewSourceFile`方法里启动了一个协程，每隔指定的时间（`kc.FileCheckFrequency`）就执行一次`SourceFile`的`run`方法，在`run`方法里所调用的主体逻辑是下面的函数：

```

func (s *sourceFile) extractFromPath() error {
    path := s.path
    statInfo, err := os.Stat(path)
    if err != nil {
        if !os.IsNotExist(err) {
            return err
        }
        // Emit an update with an empty PodList to
allow FileSource to be marked as seen
                                s.updates <-
kubelet.PodUpdate{[]*api.Pod{},           kubelet.SET,
kubelet.FileSource}
                                return fmt.Errorf("path does not exist,
ignoring")
    }

    switch {
    case statInfo.Mode().IsDir():
        pods, err := s.extractFromDir(path)
        if err != nil {
            return err
        }
        s.updates <- kubelet.PodUpdate{pods,
kubelet.SET, kubelet.FileSource}

    case statInfo.Mode().IsRegular():

```

```

        pod, err := s.extractFromFile(path)
        if err != nil {
            return err
        }

        s.updates <-
kubernetes.PodUpdate{[]*api.Pod{pod}, kubernetes.SET,
kubernetes.FileSource}

        default:
            return fmt.Errorf("path is not a directory
or file")
        }

        return nil
    }

```

看一眼上面的代码，我们就大致明白了 **Config File** 类型的 **Pod Source** 是如何工作的：它从指定的目录中加载多个 **Pod** 定义文件并转换为 **Pod** 列表或者加载单个 **Pod** 定义文件并转换为单个 **Pod**，然后生成对应的全量类型的 **PodUpdate** 事件并写入 **Channel** 中去。这里笔者也发现了代码命名的一个疏漏之处，**SourceFile** 的 **updates** 属性其实应该被命名为 **update**。其他两种 **Pod Source** 类型的代码解析就不在这里提及了。

接下来我们分析 **Merger** 对象，**PodConfig** 里的 **Merger** 对象其实是一个 **config.podStorage** 实例，它同时是 **PodConfig** 的 **pods** 属性的一个引用。**podStorage** 的源码位于 **pkg/kubelet/config/config.go** 里，其定义如下：

```
type podStorage struct {
    podLock sync.RWMutex
    // map of source name to pod name to pod reference
    pods map[string]map[string]*api.Pod
    mode PodConfigNotificationMode
    // ensures that updates are delivered in strict
order
    // on the updates channel
    updateLock sync.Mutex
    updates     chan<- kubelet.PodUpdate
    // contains the set of all sources that have sent
at least one SET
    sourcesSeenLock sync.Mutex
    sourcesSeen     util.StringSet
    // the EventRecorder to use
    recorder record.EventRecorder
}
```

我们看到podStorage的关键属性解释如下。

(1) pods: 类型是Map，存放每个Pod Source上拉过来的Pod数据，是podStorage当前保存“全量Pod”的地方。

(2) updates: 它就是PodConfig里的updates属性的一个引用。

(3) mode: 表明podStorage的Pod事件通知模式，有以下几种。

- PodConfigNotificationSnapshot: 全量快照通知模式。

- PodConfigNotificationSnapshotAndUpdates: 全量快照+更新Pod通知模式（代码中创建podStorage实例时采用的模式）。
- PodConfigNotificationIncremental: 增量通知模式。

podStorage实现的Merge接口的源码如下:

```
func (s *podStorage) Merge(source string, change
interface{}) error {
    s.updateLock.Lock()
    defer s.updateLock.Unlock()
    adds, updates, deletes := s.merge(source, change)
    // deliver update notifications
    switch s.mode {
    case PodConfigNotificationSnapshotAndUpdates:
        if len(updates.Pods) > 0 {
            s.updates <- *updates
        }
        if len(deletes.Pods) > 0 || len(adds.Pods)
> 0 {
            s.updates<-
kubelet.PodUpdate{s.MergedState().([]*api.Pod),
kubelet.SET, source}
        }
        //省略无关的Case逻辑
    }
    return nil
}
```

在上述Merge过程中，先调用内部函数merge，将Pod Soucre的Channel上发来的PodUpdate事件分解为对应的新增、更新及删除等三类PodUpdate事件，然后判断是否有更新事件，如果有，则直接写入汇总的Channel中（podStorage.updates），然后调用MergedState函数复制一份podStorage的当前全量Pod列表，以此产生一个全量的PodUpdate事件并写入汇总的Channel中，从而实现了多Pod Source Channel的“汇聚”逻辑。

分析完Merger过程以后，我们接下来看看是什么对象，以及如何消费这个汇总的Channel。在上一节提到，在kubelet进程启动的过程中调用了startKubelet方法，此方法首先启动一个协程，让kubelet处理来自PodSource的Pod Update消息，即下面这行代码：

```
go util.Forever(func() { k.Run(podCfg.Updates()) }, 0)
```

其中，PodConfig的Updates（）方法返回了前面我们所说的汇总Channel变量的一个引用，下面是kubelet的Run（updates<-chan PodUpdate）方法的代码：

```
func (kl *Kubelet) Run(updates <-chan PodUpdate) {
    if kl.logServer == nil {
        kl.logServer = http.StripPrefix("/logs/",
            http.FileServer(http.Dir("/var/log/")))
    }
    if kl.kubeClient == nil {
        glog.Warning("No api server defined - no
            node status update will be sent. ")
    }
}
```

```

    }
    // Move Kubelet to a container.
    if kl.resourceContainer != "" {
                                                err :=
util.RunInResourceContainer(kl.resourceContainer)
        if err != nil {
            glog.Warningf("Failed to move
Kubelet to container %q: %v", kl.resourceContainer, err)
        }
        glog.Infof("Running in container %q",
kl.resourceContainer)
    }
    if err := kl.imageManager.Start(); err != nil {
        kl.recorder.Eventf(kl.nodeRef, "kubeletSetupFailed",
"Failed to start ImageManager %v", err)
        glog.Errorf("Failed to start ImageManager,
images may not be garbage collected: %v", err)
    }
    if err := kl.cadvisor.Start(); err != nil {
        kl.recorder.Eventf(kl.nodeRef,
"kubeletSetupFailed", "Failed to start CAdvisor %v", err)
        glog.Errorf("Failed to start CAdvisor,
system may not be properly monitored: %v", err)
    }
    if err := kl.containerManager.Start(); err != nil {
        kl.recorder.Eventf(kl.nodeRef,
"kubeletSetupFailed", "Failed to start ContainerManager

```

```

%v", err)

                                glog.Errorf("Failed to start
ContainerManager, system may not be properly isolated: %v",
err)
    }
    if err := kl.oomWatcher.Start(kl.nodeRef); err !=
nil {
                                kl.recorder.Eventf(kl.nodeRef,
"kubeletSetupFailed", "Failed to start OOM watcher %v",
err)
                                glog.Errorf("Failed to start OOM watching:
%v", err)
    }
    go util.Until(kl.updateRuntimeUp, 5*time.Second,
util.NeverStop)
    // Run the system oom watcher forever.
    kl.statusManager.Start()
    kl.syncLoop(updates, kl)
}

```

上述代码首先启动了一个**HTTP File Server**来远程获取本节点的系统日志，接下来根据启动参数的设置来决定是否在指定的**Docker**容器中启动**kubelet**进程（如果成功，则将本进程转移到指定的容器中），然后分别启动**Image Manager**（负责**Image GC**）、**cAdvisor**（**Docker**性能监控）、**Container Manager**（**Container GC**）、**OOM Watcher**（**OOM**监测）、**Status Manager**（负责同步本节点上**Pod**的状态到**API**

Server上) 等组件, 最后进入syncLoop方法中, 无限循环调用下面的syncLoopIteration方法:

```
func (kl *Kubelet) syncLoopIteration(updates <-chan
PodUpdate, handler SyncHandler) {
    kl.syncLoopMonitor.Store(time.Now())
    if !kl.containerRuntimeUp() {
        time.Sleep(5 * time.Second)
        glog.Infof("Skipping pod synchronization,
container runtime is not up. ")
        return
    }
    if !kl.doneNetworkConfigure() {
        time.Sleep(5 * time.Second)
        glog.Infof("Skipping pod synchronization,
network is not configured")
        return
    }
    unsyncedPod := false
    podSyncTypes := make(map[types.UID]SyncPodType)
    select {
    case u, ok := <-updates:
        if !ok {
            glog.Errorf("Update channel is closed.
Exiting the sync loop.")
            return
        }
    }
```

```

        kl.podManager.UpdatePods(u, podSyncTypes)
        unsyncedPod = true
        kl.syncLoopMonitor.Store(time.Now())
    case <-time.After(kl.resyncInterval):
        glog.V(4).Infof("Periodic sync")
    }
    start := time.Now()
    // If we already caught some update, try to wait
for some short time
        // to possibly batch it with other incoming
updates.
    for unsyncedPod {
        select {
        case u := <-updates:
            kl.podManager.UpdatePods(u, podSyncTypes)
            kl.syncLoopMonitor.Store(time.Now())
        case <-time.After(5 * time.Millisecond):
            // Break the for loop.
            unsyncedPod = false
        }
    }

        pods,    mirrorPods    :=
kl.podManager.GetPodsAndMirrorMap()
        kl.syncLoopMonitor.Store(time.Now())
        if err := handler.SyncPods(pods, podSyncTypes,
mirrorPods, start); err != nil {
            glog.Errorf("Couldn't sync containers: %v",

```

```
err)
    }
    kl.syncLoopMonitor.Store(time.Now())
}
```

在上述代码中，如果从Channel中拉取到了PodUpdate事件，则先调用podManager的UpdatePods方法来确定此PodUpdate的同步类型，并将结果放入podSyncTypes这个Map中，同时为了提升处理效率，在代码中增加了持续循环拉取PodUpdate数据直到Channel为空为止（超时判断）的一段逻辑。在方法的最后，调用SyncHandler接口来完成Pod同步的具体逻辑，从而实现了PodUpdate事件的高效批处理模式。

SyncHandler在这里就是kubelet实例本身，它的SyncPods方法比较长，其主要逻辑如下。

- 将传入的全量Pod，与statusManager中当前保存的Pod集合进行对比，删除statusManager中当前已经不存在的Pod（孤儿Pod）。
- 调用kubelet的admitPods方法以过滤掉不适合本节点创建的Pod。此方法首先过滤掉状态为Failed或者Succeeded的Pod；接着过滤掉不适合本节点的Pod，比如Host Port冲突、Node Label的约束不匹配及Node的可用资源不足等情况；最后检查磁盘的使用情况，如果磁盘的可用空间不足，则过滤掉所有Pod。
- 对上述过滤后的Pod集合中的每一个Pod调用podWorkers的UpdatePod方法，而此方法内部创建了一个Pod的workUpdate事件并发布到该Pod对应的一个WorkChannel上（podWorkers.podWorkers）。

- 对于已经删除或不存在的Pod，通知podWorkers删除相关联的WorkChannel（workUpdate）。
- 对比Node当前运行中的Pod及目标Pod列表，“杀掉”多余的Pod，并且调用DockerRuntime（Docker Deamon进程）API，重新获取当前运行中的Pod列表信息。
- 清理“孤儿”Pod所遗留的PV和磁盘目录。

要真正理解Pod是怎么在Node上“落地”的，还要继续深入分析上述第3步的代码。首先我们看看对workUpdate这个结构体的定义：

```
type workUpdate struct {  
    pod *api.Pod  
    // The mirror pod of pod; nil if it does not exist.  
    mirrorPod *api.Pod  
    // Function to call when the update is complete.  
    updateCompleteFn func()  
    updateType SyncPodType  
}
```

其中的属性pod是当前要操作的Pod对象，mirrorPod则是对应的镜像Pod，下面是对它的解释：

“对于每个来自非API Server Pod Source上的Pod，kubelet都在API Server上注册一个几乎“一模一样”的Pod，这个Pod被称为mirrorPod，这样一来，就将不同的Pod Source上的Pod都“统一”到了kubelet的注册表上，从而统一了Pod生命周期的管理流程。”

`workUpdate`的`updateCompleteFn`属性是一个回调函数，`work`完成后会执行此回调函数，在上述第3步中，此函数用来计算该`work`的调度时延指标。

对于每个要同步的Pod，`podWorkers`会用一个长度为1的Channel来存放其对应的`workUpdate`，而属性`lastUndeliveredWorkUpdate`则存放最近一个待安排执行的`workUpdate`，这是因为一个Pod的前一个`workUpdate`正在执行的时候，可能会有一个新的PodUpdate事件需要处理。理解了这个过程后，再来看`podWorkers`的定义，就不难了：

```
type podWorkers struct {  
    // Protects all per worker fields.  
    podLock sync.Mutex  
    podUpdates map[types.UID]chan workUpdate  
    isWorking map[types.UID]bool  
    lastUndeliveredWorkUpdate map[types.UID]workUpdate  
    runtimeCache kubecontainer.RuntimeCache  
    syncPodFn syncPodFnType  
    recorder record.EventRecorder  
}
```

下面这个函数就是第3步里产生`workUpdate`事件并放入到`podWorkers`的对应Channel的方法的源码：

```
func (p *podWorkers) UpdatePod(pod *api.Pod, mirrorPod  
*api.Pod, updateComplete func()) {  
    uid := pod.UID
```

```

var podUpdates chan workUpdate
var exists bool
updateType := SyncPodUpdate
p.podLock.Lock()
defer p.podLock.Unlock()
    if podUpdates, exists = p.podUpdates[uid]; !exists
{
    podUpdates = make(chan workUpdate, 1)
    p.podUpdates[uid] = podUpdates
    updateType = SyncPodCreate
    go func() {
        defer util.HandleCrash()
        p.managePodLoop(podUpdates)
    }()
}
if !p.isWorking[pod.UID] {
    p.isWorking[pod.UID] = true
    podUpdates <- workUpdate{
        pod: pod,
        mirrorPod: mirrorPod,
        updateCompleteFn: updateComplete,
        updateType: updateType,
    }
} else {
    p.lastUndeliveredWorkUpdate[pod.UID] =
workUpdate{
        pod: pod,

```

```

        mirrorPod:      mirrorPod,
        updateCompleteFn: updateComplete,
        updateType:      updateType,
    }
}
}

```

上面的代码会调用 podWorkers 的 managePodLoop 方法来处理 podUpdates 队列，这里主要是获取必要的参数，最终处理又转手交给 syncPodFn 方法去处理。下面是 managePodLoop 的源码：

```

func (p *podWorkers) managePodLoop(podUpdates <-chan
workUpdate) {
    var minRuntimeCacheTime time.Time
    for newWork := range podUpdates {
        func() {
            defer p.checkForUpdates(newWork.pod.UID,
newWork.updateCompleteFn)

                                if err :=
p.runtimeCache.ForceUpdateIfOlder(minRuntimeCacheTime); err
!= nil {
                glog.Errorf("Error updating the container runtime
cache: %v", err)

                                return
                            }
        pods, err := p.runtimeCache.GetPods()
        if err != nil {

```

```

                                glog.Errorf("Error getting pods while
syncing pod: %v", err)

                                return
                        }

                        err = p.syncPodFn(newWork.pod,
newWork.mirrorPod,

kubecontainer.Pods(pods).FindPodByID(newWork.pod.UID),
newWork.updateType)

                                if err != nil {
                                        glog.Errorf("Error syncing pod %s, skipping: %v",
newWork.pod.UID, err)
                                        p.recorder.Eventf(newWork.pod, "failedSync", "Error
syncing pod, skipping: %v", err)
                                        return
                                }

                                minRuntimeCacheTime = time.Now()
                                newWork.updateCompleteFn()

                        }()
                }
        }

```

追踪podWorkers的构造函数调用过程，可以发现syncPodFn函数其实就是kubelet的syncPod方法，这个方法的代码量有点儿多，主要逻辑如下。

(1) 根据系统配置中的权限控制，检查Pod是否有权在本节点运行，这些权限包括Pod是否有权使用HostNetwork（还记得之前分析的代码么？由Pod Source类型决定）、Pod中的容器是否被授权以特权模式启动（privileged mode）等，如果未被授权，则删除当前运行中的旧版本的Pod实例并返回错误信息。

(2) 创建Pod相关的工作目录、PV存放目录、Plugin插件目录，这些目录都以Pod的UID为上一级目录。

(3) 如果Pod有PV定义，则针对每个PV执行目录的mount操作。

(4) 如果是SyncPodUpdate类型的Pod，则从DockerRuntime的API接口查询获取Pod及相关容器的最新状态信息。

(5) 如果Pod有imagePullSecrets属性，则在API Server上获取对应的Secret。

(6) 调用Container Runtime的API接口方法SyncPod，实现Pod“真正同步”的逻辑。

(7) 如果Pod Source不来自API Server，则继续处理其关联的mirrorPod。

- 如果mirrorPod跟当前Pod的定义不匹配，则它会被删除。
- 如果mirrorPod还不存在（比如新创建的Pod），则会在API Server上新建一个。

Kubernetes中Container Runtime的默认实现是Docker，对应类是dockertools.DockerManager，其源码位于

kg/kubelet/dockertools/manager.go 里，在上述 kubelet.syncPod 方法中所调用的 DockerManager 的 SyncPod 方法实现了下面的逻辑。

- 判断一个 Pod 实例的哪些组成部分需要重启：包括 Pod 的 infra 容器是否发生变化（如网络模式、Pod 里运行的各个容器的端口是否发生变化）；Pod 里运行的容器是否发生变化；用 Probe 检测容器的状态以确定容器是否异常等。
- 根据 Pod 实例重启结果的判断，如果需要重启 Pod 的 infra 容器，则先 Kill Pod 然后启动 Pod 的 infra 容器，设定好网络，最后启动 Pod 里的所有 Container；否则就先 Kill 那些需要重启的 Container，然后重新启动它们。注意，如果是新创建的 Pod，则因为找不到 Node 上对应的 Pod 的 infra 容器，所以会被当作重启 Pod 的 infra 容器的逻辑来实现创建过程。

DockerManager 创建 Pod 的 infra 容器的逻辑在 createPodInfraContainer 方法里，大体逻辑如下。

- 如果 Pod 的网络不是 HostNetwork 模式，则搜集 Pod 所有容器的 Port 作为 infra 容器所要暴露的 Port 列表。
- 如果 infra 容器的 Image 目前不存在，则尝试拉取 Image。
- 创建 infra 的 Container 对象并且启动 runContainerInPod 方法。
- 如果容器定义有 Lifecycle，并且 PostStart 回调方法被设置了，就会触发此方法的调用，如果调用失败则 Kill 容器并返回。
- 创建一个软连接文件指向容器的日志文件，此软连接文件名包括 Pod 的名称、容器的名称及容器的 ID，这样的目的是让 Elasticsearch 这样的搜索技术容易索引和定位 Pod 日志。
- 如果此容器是 Podinfra 容器，则设置其 OOM 参数低于标准值，使得它比其他容器具备更强的“抗灾”能力。

- 修改Docker生成的容器的resolv.conf文件，增加ndots参数并默认设置为5，这是因为Kubernetes默认假设的域名分割长度是5，例如_dns._udp.kube-dns.default.svc。

上述逻辑中所调用的runContainerInPod是DockerManager的核心方法之一，不管是创建Pod的infra容器还是Pod里的其他容器，都会通过此方法使得容器被创建和运行。以下是其主要逻辑。

- 生成Container必要的环境变量和参数，比如ENV环境变量、Volume Mounts信息、端口映射信息、DNS服务器信息、容器的日志目录、parent cgGroup等。
- 调用runContainer方法完成Docker Container实例的创建过程，简单地说，就是完成Docker create container命令行所需的各种参数的构造过程，并通过程序来调用执行。
- 构造HostConfig对象，主要参数有目录映射、端口映射等、cgGroup的设定等，简单地说，就是完成了Docker start container命令行所需的必要参数的构造过程，并通过程序来调用执行。

在上述逻辑中，runContainer与startContainer的具体实现都是靠DockerManager中的dockerClient对象完成的，它实现了DockerInterface接口，dockerClient的创建过程在pkg/kubelet/dockertools/docker.go里，下面是这段代码：

```
func ConnectToDockerOrDie(dockerEndpoint string)
DockerInterface {
    if dockerEndpoint == "fake://" {
        return &FakeDockerClient{
```

VersionInfo:

```

docker.Env{"ApiVersion=1.18"},
        }
    }

    client, err :=
docker.NewClient(getDockerEndpoint(dockerEndpoint))
    if err != nil {
        glog.Fatalf("Couldn't connect to docker:
%v", err)
    }
    return client
}

```

这里的`dockerEndpoint`是本节点上的Docker Deamon进程的访问地址，默认是`unix: ///var/run/docker.sock`，在上述代码中使用了来自开源项目<https://github.com/fsouza/go-dockerclient>提供的Docker Client，它也是Go语言实现的一个用HTTP访问Docker Deamon提供的标准API的客户端框架。

我们来看看 `dockerClient` 创建容器的具体代码（`CreateContainer`）：

```

func (c *Client) CreateContainer(opts
CreateContainerOptions) (*Container, error) {
    path := "/containers/create " + queryString(opts)
    body, status, err := c.do(
        "POST",
        path,

```



```

doOptions{
    data: struct {
        *Config
        HostConfig *HostConfig
`json:"HostConfig,omitempty" yaml:"HostConfig,omitempty"`
    }{
        opts.Config,
        opts.HostConfig,
    },
},
)
if status == http.StatusNotFound {
    return nil, ErrNoSuchImage
}
if err != nil {
    return nil, err
}
var container Container
err = json.Unmarshal(body, &container)
if err != nil {
    return nil, err
}
container.Name = opts.Name
return &container, nil
}

```

上述代码其实就是通过调用标准的Docker RestAPI来实现功能的，我们进入docker.Client的do方法里可以看到更多详情，例如输入参数转换为JSON格式的数据、DockerAPI版本检查及异常处理等逻辑，最有趣的是：在dockerEndpoint是unix套接字的情况下，会先建立套接字连接，然后在这个连接上创建HTTP连接。

至此，我们分析了kubelet创建和同步Pod实例的整个流程，简单总结如下。

- 汇总：先将多个Pod Source上过来的PodUpdate事件汇聚到一个总的Channel上去。
- 初审：分析并过滤掉不符合本节点的PodUpdate事件，对满足条件的PodUpdate则生成一个workUpdate事件，交给podWorkers处理。
- 接待：podWorkers对每个Pod的workUpdate事件排队，并且负责更新Cache中的Pod状态，而把具体的任务转给kubelet去处理（syncPod方法）。
- 终审：kubelet对符合条件的Pod进一步审查，例如检查Pod是否有权在本节点运行，对符合审查的Pod开始着手准备工作，包括目录创建、PV创建、Image获取、处理Mirror Pod问题等，然后把“皮球”踢给了DockerManager。
- 落地：任务抵达DockerManager之后，DockerManager尽心尽责地分析每个Pod的情况，以决定这个Pod究竟是新建、完全重启还是部分更新的。给出分析结果以后，剩下的就是dockerClient的工作了。

好复杂的设计！原来非业务流程的代码理解起来也会如此折磨人，真心不知道谷歌当初是怎么设计和实现它的，目测国内P8水平的一帮大牛们天天加班到9点钟，也难以交付这样的Code。

在继续下面的分析之前，留一个小小的思考给聪明的读者：Pod Source上发来的Pod删除的事件，是在哪里处理的？

接下来我们继续分析kubelet进程的另外一个重要功能是如何实现的，即定期同步Pod状态信息到API Server上。先来看看Pod状态的数据结构定义：

```
type PodStatus struct {
    Phase          PodPhase          `json:"phase,omitempty"`
    Conditions     []PodCondition
    `json:"conditions,omitempty"`
    Message string `json:"message,omitempty"`
    Reason string `json:"reason,omitempty"`
    HostIP string `json:"hostIP,omitempty"`
    PodIP  string `json:"podIP,omitempty"`
    StartTime *util.Time `json:"startTime,omitempty"`
    ContainerStatuses []ContainerStatus
}

// PodStatusResult is a wrapper for PodStatus returned
by kubelet that can be encode/decoded

type PodStatusResult struct {
    TypeMeta    `json:",inline"`
    ObjectMeta `json:"metadata,omitempty"`
    Status PodStatus `json:"status,omitempty"`
}

```

Pod的状态（Phase）有5种：运行中（PodRunning）、等待中（PodPending）、正常终止（PodSucceeded）、异常停止（PodFailed）及未知状态（PodUnknown），最后一种状态很可能是由于Pod所在主机的通信问题导致的。从上面的定义可以看到Pod的状态同时包括它里面运行的Container的状态，另外给出了导致当前状态的原因说明、Pod的启动时间等信息。PodStatusResult则是Kubernet API Server提供的Pod Status API接口中用到的Wrapper类。

通过之前的代码研读，我们发现在Kubernetes中大量使用了Channel和协程机制来完成数据的高效传递和处理工作，在kubelet中更是大量使用了这一机制，实现PodStatus上报的kubelet.statusManager也是如此，它用一个Map（podStatuses）保存了当前kubelet中所有Pod实例的当前状态，并且声明了一个Channel（podStatusChannel）来存放Pod状态同步的更新请求（podStatuses），Pod在本地实例化和同步的过程中会引发Pod状态的变化，这些变化被封装为podStatusSyncRequest放入Channel中，然后被异步上报到API Server，这就是statusManager的运行机制。

下面是statusManager的SetPodStatus方法，先比较缓存的状态信息，如果状态发生变化，则触发Pod状态，生成podStatusSyncRequest并放到队列中等待上报：

```
func (s *statusManager) SetPodStatus(pod *api.Pod,
status api.PodStatus) {
    podFullName := kubecontainer.GetPodFullName(pod)
    s.podStatusesLock.Lock()
    defer s.podStatusesLock.Unlock()
```

```

        oldStatus, found := s.podStatuses[podFullName]
        // ensure that the start time does not change
across updates.
        if found && oldStatus.StartTime != nil {
            status.StartTime = oldStatus.StartTime
        }
        if status.StartTime.IsZero() {
            if pod.Status.StartTime.IsZero() {
                // the pod did not have a
previously recorded value so set to now
                now := util.Now()
                status.StartTime = &now
            } else {
                status.StartTime =
pod.Status.StartTime
            }
        }
        if !found || !isStatusEqual(&oldStatus, &status) {
            s.podStatuses[podFullName] = status
            s.podStatusChannel <-
podStatusSyncRequest{pod, status}
        } else {
            glog.V(3).Infof("Ignoring same status for
pod %q, status: %v", kubeletUtil.FormatPodName(pod),
status)
        }
    }
}

```

下面是在Pod实例化的过程中，kublet过滤掉不合适本节点Pod所调用的上述方法的代码，类似的调用还有不少：

```
func (kl *Kubelet) handleNotFittingPods(pods
[]*api.Pod) []*api.Pod {
    fitting, notFitting := checkHostPortConflicts(pods)
    for _, pod := range notFitting {
        reason := "HostPortConflict"
        kl.recorder.Eventf(pod, reason, "Cannot
start the pod due to host port conflict.")
        kl.statusManager.SetPodStatus(pod,
api.PodStatus{
            Phase:    api.PodFailed,
            Reason:    reason,
            Message:    "Pod cannot be started due
to host port conflict"})
    }

    fitting, notFitting =
kl.checkNodeSelectorMatching(fitting)
    for _, pod := range notFitting {
        reason := "NodeSelectorMismatching"
        kl.recorder.Eventf(pod, reason, "Cannot
start the pod due to node selector mismatch.")
        kl.statusManager.SetPodStatus(pod,
api.PodStatus{
            Phase:    api.PodFailed,
```

```

                Reason:  reason,
                Message: "Pod cannot be started due
to node selector mismatch"})
        }

        fitting,    notFitting    =
k1.checkCapacityExceeded(fitting)
        for _, pod := range notFitting {
                reason := "CapacityExceeded"
                k1.recorder.Eventf(pod, reason, "Cannot
start the pod due to exceeded capacity.")
                k1.statusManager.SetPodStatus(pod,
api.PodStatus{
                        Phase:  api.PodFailed,
                        Reason:  reason,
                        Message: "Pod cannot be started due
to exceeded capacity"})
        }
        return fitting
}

```

最后，我们看看statusManager是怎么把Channel的数据上报到API Server上的，这是通过Start方法开启一个协程无限循环执行syncBatch方法来实现的，下面是syncBatch的代码：

```

func (s *statusManager) syncBatch() error {
    syncRequest := <-s.podStatusChannel
    pod := syncRequest.pod

```

```

    podFullName := kubecontainer.GetPodFullName(pod)
    status := syncRequest.status

    var err error
    statusPod := &api.Pod{
        ObjectMeta: pod.ObjectMeta,
    }

    statusPod, err =
s.kubeClient.Pods(statusPod.Namespace).Get(statusPod.Name)
    if err == nil {
        statusPod.Status = status

        _, err =
s.kubeClient.Pods(pod.Namespace).UpdateStatus(statusPod)
        // TODO: handle conflict as a retry, make
that easier too.

        if err == nil {
            glog.V(3).Infof("Status for pod %q
updated successfully", kubeletUtil.FormatPodName(pod))
            return nil
        }
    }

    go s.DeletePodStatus(podFullName)
    return fmt.Errorf("error updating status for pod
%q: %v", kubeletUtil.FormatPodName(pod), err)
}

```

这段代码首先从Channel中拉取一个syncRequest，然后调用API Server接口来获取最新的Pod信息，如果成功，则继续调用API Server的UpdateStatus接口更新Pod状态，如果调用失败则删除缓存的Pod状态，这将触发kubelet重新计算Pod状态并再次尝试更新。

说完了Pod流程，我们接下来再一起深入分析Kubernetes中的容器探针（Probe）的实现机制。我们知道，容器正常不代表里面运行的业务进程能正常工作，比如程序还没初始化好，或者配置文件错误导致无法正常服务，还有诸如数据库连接爆满导致服务异常等各种意外情况都有可能发生，面对这类问题，cAdvisor就束手无策了，所以kubelet引入了容器探针技术，容器探针按照作用划分为以下两种。

- **ReadinessProbe**: 用来探测容器中的用户服务进程是否处于“可服务状态”，此探针不会导致容器被停止或重启，而是导致此容器上的服务被标识为不可用，Kubernetes不会发送请求到不可用的容器上，直到它们可用为止。
- **LivenessProbe**: 用来探测容器服务是否处于“存活状态”，如果服务当前被检测为Dead，则会导致容器重启事件发生。

下面是探针相关的结构定义：

```
type Probe struct {
    Handler
    InitialDelaySeconds    int64
    TimeoutSeconds    int64
}

type Handler struct {
    // One and only one of the following should be
```

```
specified.  
  
    Exec *ExecAction  
  
    HTTPGet *HTTPGetAction  
  
    TCPSocket *TCPSocketAction  
  
}
```

从上面的定义来看，探针可以通过执行容器中的一个命令、发起一个指向容器内部的HTTP Get请求或者TCP连接来确定容器内部是否正常工作。

上面的代码属于API包中的一部分，只是用来描述和存储容器上的探针定义，而真正的探针实现代码则位于pkg/kubelet/prober/prober.go里，下面是对prober.Probe的定义：

```
type Prober interface {  
    Probe(pod *api.Pod, status api.PodStatus, container  
api.Container, containerID string, createdAt int64)  
(probe.Result, error)  
}
```

上述接口方法表示对一个Container发起探测并返回其结果。prober.Probe的实现类为prober.prober，其结构定义如下：

```
type prober struct {  
    exec    execprobe.ExecProber  
    http    httpprobe.HTTPProber  
    tcp     tcpprobe.TCPProber
```

```

runner kubecontainer.ContainerCommandRunner
readinessManager *kubecontainer.ReadinessManager
refManager       *kubecontainer.RefManager
recorder         record.EventRecorder
}

```

其中exec、http、tcp三个变量分别对应三种探测类型的“探头”，它们已经各自实现了相应的逻辑。比如下面这段代码是HTTP探头的核心逻辑，即连接一个URL发起GET请求：

```

func DoHTTPProbe(url *url.URL, client
HTTPGetInterface) (probe.Result, string, error) {
    res, err := client.Get(url.String())
    if err != nil {
        // Convert errors into failures to catch
        timeouts.

        return probe.Failure, err.Error(), nil
    }
    defer res.Body.Close()
    b, err := ioutil.ReadAll(res.Body)
    if err != nil {
        return probe.Failure, "", err
    }
    body := string(b)

    if res.StatusCode >= http.StatusOK &&
res.StatusCode < http.StatusBadRequest {
        glog.V(4).Infof("Probe succeeded for %s,

```

```
Response: %v", url.String(), *res)
        return probe.Success, body, nil
    }
    glog.V(4).Infof("Probe failed for %s, Response:
    %v", url.String(), *res)
    return probe.Failure, body, nil
}
```

prober.prober中的runner则是exec探头的执行器，因为后者需要在被检测的容器中执行一个cmd命令：

```
func (p *prober) newExecInContainer(pod *api.Pod,
container api.Container, containerID string, cmd []string)
exec.Cmd {
    return execInContainer(func() ([]byte, error) {
        return p.runner.RunInContainer(containerID,
cmd)
    })
}
```

实际上p.runner就是之前我们分析过的DockerManager，下面是RunInContainer的源码：

```
func (dm *DockerManager) RunInContainer(containerID
string, cmd []string) ([]byte, error) {
    // If native exec support does not exist in the
local docker daemon use nsinit.
```

```

        useNativeExec, err := dm.nativeExecSupportExists()
        if err != nil {
            return nil, err
        }
        if !useNativeExec {
            glog.V(2).Infof("Using nsinit to run the
command %+v inside container %s", cmd, containerID)

            return
dm.runInContainerUsingNsinit(containerID, cmd)
        }
        glog.V(2).Infof("Using docker native exec to run
cmd %+v inside container %s", cmd, containerID)
        createOpts := docker.CreateExecOptions{
            Container:    containerID,
            Cmd:          cmd,
            AttachStdin:   false,
            AttachStdout:  true,
            AttachStderr:  true,
            Tty:           false,
        }
        execObj, err := dm.client.CreateExec(createOpts)
        if err != nil {
            return nil, fmt.Errorf("failed to run in
container - Exec setup failed - %v", err)
        }
        var buf bytes.Buffer
        startOpts := docker.StartExecOptions{

```

```

        Detach:      false,
        Tty:         false,
        OutputStream: &buf,
        ErrorStream:  &buf,
        RawTerminal:  false,
    }
    err = dm.client.StartExec(execObj.ID, startOpts)
    if err != nil {
        glog.V(2).Infof("StartExec With error: %v",
err)

        return nil, err
    }
    ticker := time.NewTicker(2 * time.Second)
    defer ticker.Stop()
    for {
                                                inspect,    err2    :=
dm.client.InspectExec(execObj.ID)
        if err2 != nil {
            glog.V(2).Infof("InspectExec %s failed
with error: %+v", execObj.ID, err2)
            return buf.Bytes(), err2
        }
        if !inspect.Running {
            if inspect.ExitCode != 0 {
                glog.V(2).Infof("InspectExec %s exit
with result %+v", execObj.ID, inspect)
                err = &dockerExitError{inspect}
            }
        }
    }

```

```

        }
        break
    }
    <-ticker.C
}

return buf.Bytes(), err
}

```

Docker自1.3版本开始支持使用Exec指令（以及API调用）在容器内执行一个命令，我们看看上述过程中使用的dm.client.CreateExec方法是如何实现的：

```

func (c *Client) CreateExec(opts CreateExecOptions)
(*Exec, error) {
    path := fmt.Sprintf("/containers/%s/exec",
opts.Container)
    body, status, err := c.do("POST", path,
doOptions{data: opts})
    if status == http.StatusNotFound {
        return nil, &NoSuchContainer{ID:
opts.Container}
    }
    if err != nil {
        return nil, err
    }
    var exec Exec

```

```
    err = json.Unmarshal(body, &exec)
    if err != nil {
        return nil, err
    }
    return &exec, nil
}
```

我们看到，这是标准的Docker API的调用方式，跟之前看到的创建容器的调用代码很相似。现在我们再回头看看prober.prober是怎么执行ReadinessProbe/LivenessProbe的检测逻辑的：

```
func (pb *prober) Probe(pod *api.Pod, status
api.PodStatus, container api.Container, containerID string,
createdAt int64) (probe.Result, error) {
    pb.probeReadiness(pod, status, container,
containerID, createdAt)
    return pb.probeLiveness(pod, status, container,
containerID, createdAt)
}
```

这段代码先调用容器的ReadinessProbe进行检测，并且在readinessManager组件中记录容器的Readiness状态，随后调用容器的LivenessProbe进行检测，并返回容器的状态，在检测过程中如果发现状态为失败或者异常状态，则会连续检测3次：

```
func (pb *prober) runProbeWithRetries(p *api.Probe,
pod *api.Pod, status api.PodStatus, container
```



```

api.Container,    containerID    string,    retries    int)
(probe.Result, string, error) {
    var err error
    var result probe.Result
    var output string
    for i := 0; i < retries; i++ {
        result, output, err = pb.runProbe(p, pod,
status, container, containerID)
        if result == probe.Success {
            return probe.Success, output, nil
        }
    }
    return result, output, err
}

```

比较意外的是`prober.prober`探针检测容器状态的方法目前只在一处被调用到，位于方法`DockerManager.computePodContainerChanges`里：

```

    result, err := dm.prober.Probe(pod, podStatus,
container, string(c.ID), c.Created)
    if err != nil {
        // TODO(vmarmol): examine this
logic.

        glog.V(2).Infof("probe no-error:
%q", container.Name)

        containersToKeep[containerID] =
index

```

```

        continue
    }
    if result == probe.Success {
        glog.V(4).Infof("probe success:
%q", container.Name)

        containersToKeep[containerID] =
index

        continue
    }

    glog.Infof("pod %q container %q is
unhealthy (probe result: %v), it will be killed and re-
created.", podFullName, container.Name, result)
    containersToStart[index] = empty{}
}

```

只有没有发生任何变化的**Pod**才会执行一次探针检测，若检测状态为失败，则会导致重启事件发生。

本节最后，我们再来简单分析下**kubelet**中的**Kubelet Server**的实现机制，下面是**kubelet**进程启动过程中启动**Kubelet Server**的源码入口：

```

// start the kubelet server
if kc.EnableServer {
    go util.Forever(func() {
        k.ListenAndServe(net.IP(kc.Address),
kc.Port, kc.TLSOptions, kc.EnableDebuggingHandlers)

```

```
    }, 0)
}
```

在上述代码调用的过程中，创建了一个类型为`kubelet.Server`的HTTP Server并在本地监听：

```
    handler := NewServer(host, enableDebuggingHandlers)
    s := &http.Server{
                                                Addr:
net.JoinHostPort(address.String(),
strconv.FormatUint(uint64(port), 10)),
        Handler:      &handler,
        ReadTimeout:   5 * time.Minute,
        WriteTimeout:  5 * time.Minute,
        MaxHeaderBytes: 1 << 20,
    }
    if tlsOptions != nil {
        s.TLSConfig = tlsOptions.Config

glog.Fatal(s.ListenAndServeTLS(tlsOptions.CertFile,
tlsOptions.KeyFile))
    } else {
        glog.Fatal(s.ListenAndServe())
    }
```

在`kubelet.Server`的构造函数里加载如下HTTP Handler:

```
func (s *Server) InstallDefaultHandlers() {
    healthz.InstallHandler(s.mux,
        healthz.PingHealthz,
        healthz.NamedCheck("docker",
s.dockerHealthCheck),
        healthz.NamedCheck("hostname",
s.hostnameHealthCheck),
        healthz.NamedCheck("syncloop",
s.syncLoopHealthCheck),
    )
    s.mux.HandleFunc("/pods", s.handlePods)
    s.mux.HandleFunc("/stats/", s.handleStats)
    s.mux.HandleFunc("/spec/", s.handleSpec)
}
```

上述Handler分为两组：首先是健康检查，包括kublet进程自身的心跳检查、Docker进程的健康检查、kublet所在主机名检测、Pod同步的健康检查等；然后是获取当前节点上运行期信息的接口，例如获取当前节点上的Pod列表、统计信息等。下面是hostnameHealthCheck的实现逻辑，它检查Pod两次同步之间的时延，而这个时延则在之前提到的kublet的syncLoopIteration方法中进行更新：

```
func (s *Server) syncLoopHealthCheck(req
*http.Request) error {
    duration := s.host.ResyncInterval() * 2
    minDuration := time.Minute * 5
    if duration < minDuration {
```

```

        duration = minDuration
    }
    enterLoopTime := s.host.LatestLoopEntryTime()
    if !enterLoopTime.IsZero() &&
time.Now().After(enterLoopTime.Add(duration)) {
        return fmt.Errorf("Sync Loop took longer
than expected.")
    }
    return nil
}

```

handlePods的API则从kubelet中获取当前“绑定”到本节点的所有Pod的信息并返回:

```

func (s *Server) handlePods(w http.ResponseWriter, req
*http.Request) {
    pods := s.host.GetPods()
    data, err := encodePods(pods)
    if err != nil {
        s.error(w, err)
        return
    }
    w.Header().Add("Content-type", "application/json")
    w.Write(data)
}

```

如果kubelet运行在Debug模式, 则加载更多的HTTP Handler:

```
func (s *Server) InstallDebuggingHandlers() {
    s.mux.HandleFunc("/run/", s.handleRun)
    s.mux.HandleFunc("/exec/", s.handleExec)
    s.mux.HandleFunc("/portForward/",
s.handlePortForward)

    s.mux.HandleFunc("/logs/", s.handleLogs)
    s.mux.HandleFunc("/containerLogs/",
s.handleContainerLogs)
    s.mux.Handle("/metrics", prometheus.Handler())
    // The /runningpods endpoint is used for testing
only.
    s.mux.HandleFunc("/runningpods",
s.handleRunningPods)

    s.mux.HandleFunc("/debug/pprof/", pprof.Index)
    s.mux.HandleFunc("/debug/pprof/profile",
pprof.Profile)
    s.mux.HandleFunc("/debug/pprof/symbol",
pprof.Symbol)
}
```

这些HTTPHandler的实现并不复杂，所以在这里就不再一一介绍了。

6.5.3 设计总结

在研读kubernetes源码的过程中，你经常会有“山穷水尽疑无路，柳暗花明又一村”的感觉，是因为在它的设计中大量运用了Channel这种异步消息机制，加之为了测试的方便，又将很多重要的处理函数做成接口类，只有找到并分析这些接口的具体实现类，才能明白整个流程。这对于习惯了面向对象语言的程序员而言，有一种一夜回到解放前的感觉。

因为kubernetes的功能比较多，所以我们在此仅以Pod同步的主流程为例，进行一个设计总结，图6.8是kubernetes主流程相关的设计示意图，为了更加清晰地展示整个流程，我们特意将kubernetes Kernel、Docker System与其他部分分离开来，并且省略了部分非核心对象和数据结构。

首先，config.PodConfig创建一个或多个Pod Source，在默认情况下创建的是API source，它并没有创建新的数据结构，而是使用之前介绍的cache.Reflector结合cache.UndeltaStore，从Kubernetes API Server上拉取Pod数据放入内部的Channel上，而内部的Channel收到Pod数据后会调用podStorage的Merge方法实现多个Channel数据的合并，产生kubernetes.PodUpdate消息并写入PodConfig的汇总Channel上，随后PodUpdate消息进入kubernetes Kernel中进行下一步处理。

kubernetes.kubernetes的syncLoop方法监听PodConfig的汇总Channel，过滤掉不合适的PodUpdate并把符合条件的放入SyncPods方法中，最终为

每个符合条件的 Pod 产生一个 `kubelet.workUpdate` 事件并放入 `podWorkers` 的内部工作队列上，随后调用 `podWorkers` 的 `managePodLoop` 方法进行处理。 `podWorkers` 在处理流程中调用了 `DockerManager` 的 `SyncPod` 方法，由此 `DockerManager` 接班，在进行了必要的 Pod 周边操作后，对于需要重启或者更新的容器， `DockerManager` 则交给 `docker.Client` 对象去执行具体的动作，后者通过调用 `Dockers Engine` 的 `API Service` 来实现具体功能。

在 Pod 同步的过程中会产生 Pod 状态的变更和同步问题，这些是由 `kubelet.statusManager` 实现的，它在内部也采用了 `Channel` 的设计方式。

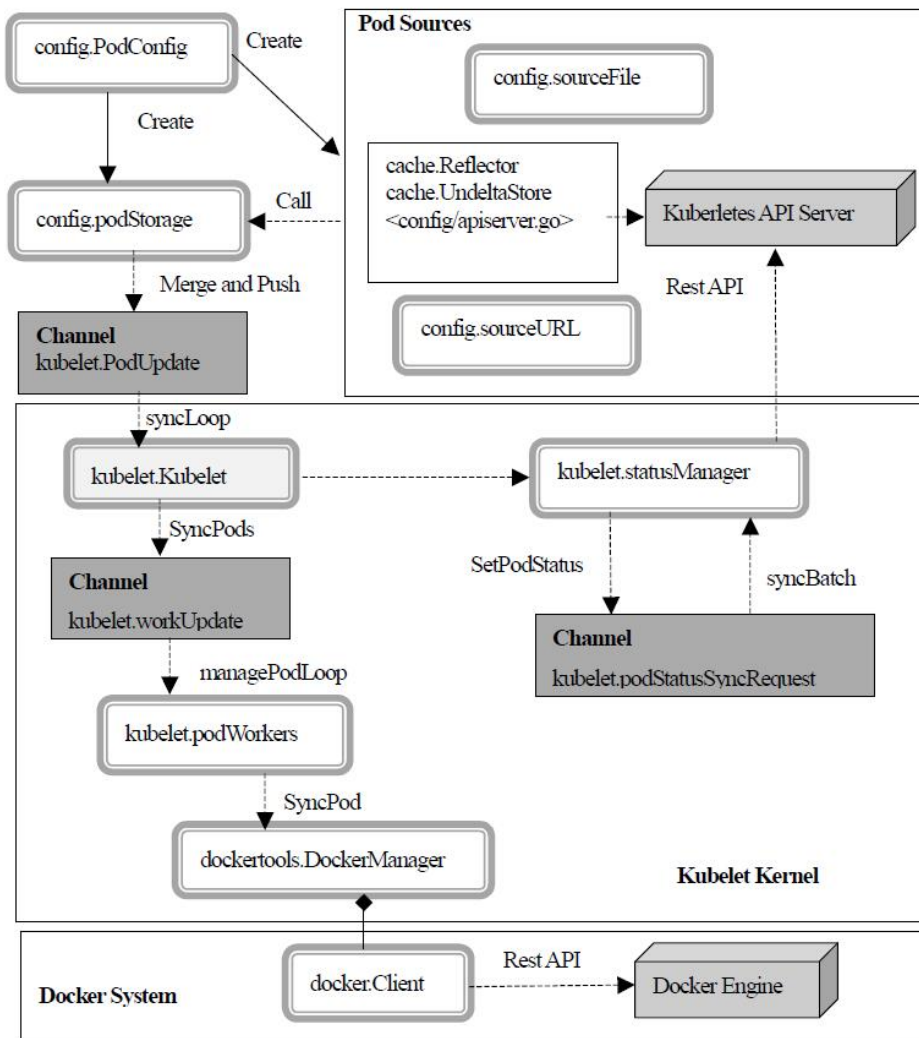


图6.8 kubelet主流程相关的设计示意图

6.6 kube-proxy进程源码分析

kube-proxy是运行在**Minion**节点上的另外一个重要的守护进程，你可以把它当作一个**HAProxy**，它充当了**Kubernetes**中**Service**的负载均衡器和服务代理的角色。下面我们分别对其启动过程、关键代码分析及设计总结等方面进行深入分析和讲解。

6.6.1 进程启动过程

kube-proxy进程的入口类源码位置如下:

```
github.com/GoogleCloudPlatform/kubernetes/cmd/kube-proxy/proxy.go
```

入口main () 函数的逻辑如下:

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())
    s := app.NewProxyServer()
    s.AddFlags(pflag.CommandLine)

    util.InitFlags()
    util.InitLogs()
    defer util.FlushLogs()

    verflag.PrintAndExitIfRequested()

    if err := s.Run(pflag.CommandLine.Args()); err !=
nil {
        fmt.Fprintf(os.Stderr, "%v\n", err)
        os.Exit(1)
    }
}
```

```
    }  
}
```

上述代码构造了一个**ProxyServer**，然后调用它的**Run**方法启动运行。首先我们看看**NewProxyServer**的代码：

```
func NewProxyServer() *ProxyServer {  
    return &ProxyServer{  
  
                                BindAddress:  
util.IP(net.ParseIP("0.0.0.0")),  
                                HealthzPort:      10249,  
                                HealthzBindAddress:  
util.IP(net.ParseIP("127.0.0.1")),  
                                OOMScoreAdj:      -899,  
                                ResourceContainer: "/kube-proxy",  
    }  
}
```

在上述代码中，**ProxyServer**绑定本地所有IP（0.0.0.0）对外提供代理服务，而提供健康检查的**HTTP Server**则默认绑定本地的回环IP，说明后者仅用于在本节点上访问，如果需要开发管理系统进行远程管理，则可以设置参数**healthz-bind-address**为0.0.0.0来达到目的。另外，从代码中看，**ProxyServer**还有一个重要属性可以调整：**PortRange**（对应命令行参数为**proxy-port-range**），它用来限定**ProxyServer**使用哪些本地端口作为代理端口，默认是随机选择。

ProxyServer的**Run**方法流程如下。

- 设置本进程的OOM参数OOMScoreAdj，保证系统OOM时， kube-proxy不会首先被系统删除，这是因为kube-proxy与kubelet进程一样，比节点上的Pod进程更重要。
- 让自己的进程运行在指定的Linux Container中，这个Container的名字来自ProxyServer.ResourceContainer，如上所述，默认为/kube-proxy，比较重要的一点是这个Container具备所有设备的访问权。
- 创建ServiceConfig与EndpointsConfig，它们与之前kubelet中的PodConfig的作用和实现机制有点像，分别负责监听和拉取API Server上Service与Service Endpoints的信息，并通知给注册到它们上的Listener接口进行处理。
- 创建一个 round-robin 轮询机制的 load balancer（LoadBalancerRR），它用来实现Service的负载均衡转发逻辑，它也是前面创建的EndpointsConfig的一个Listener。
- 创建一个Proxier，它负责建立和维护Service的本地代理Socket，它也是前面创建的ServiceConfig的一个Listener。
- 创建一个config.SourceAPI，并启动两个协程，通过Kubernetes Client来拉取Kubernetes API Server上的Service与Endpoint数据，然后分别写入之前定义的ServiceConfig与EndpointsConfig的Channel上，从而触发整个流程的驱动。
- 本地绑定健康检查的HTTP Server提供服务。
- 进入Proxier的SyncLoop方法里，该方法周期性地检查Iptables是否设置正常、服务的Portal是否正常开启，以及清除load balancer上的过期会话。

从启动流程看， kube-proxy进程的参数比较少，它所做的事情也是比较单一的，没有kubelet进程那么复杂，在下一节我们会深入分析其关键代码。

6.6.2 关键代码分析

从上一节 kube-proxy 的启动流程来看，它跟 kubelet 有相似的地方，即都会从 Kubernetes API Server 拉取相关的资源数据并在本地节点上完成“深加工”，其拉取资源的做法，第一眼看上去与 kubelet 相似，但实际上有稍微不同的实现思路，这说明作者另有其人。

由于 ServiceConfig 与 EndpointsConfig 实现机制是完全一样的，只不过拉取的资源不同，所以我们这里仅对前者做深入分析。首先从 ServiceConfig 结构体开始：

```
type ServiceConfig struct {  
    mux      *config.Mux  
    bcaster  *config.Broadcaster  
    store    *serviceStore  
}
```

ServiceConfig 也使用了 mux（config.Mux），它是一个多 Channel 的多路合并器，之前 kubelet 的 PodConfig 也用到了它。下面是 ServiceConfig 的构造函数：

```
func NewServiceConfig() *ServiceConfig {  
    updates := make(chan struct{})  
    store := &serviceStore{updates: updates, services:  
make(map[string]map[types.NamespaceName]api.Service)}
```

```

types.NamespacedName{value.Namespace, value.Name}
        services[name] = value
    }
    case REMOVE:
        glog.V(4).Infof("Removing a service %+v",
update)
        for _, value := range update.Services {
            name :=
types.NamespacedName{value.Namespace, value.Name}
            delete(services, name)
        }
    case SET:
        glog.V(4).Infof("Setting services %+v",
update)
        // Clear the old map entries by just
creating a new map
        services =
make(map[types.NamespacedName]api.Service)
        for _, value := range update.Services {
            name :=
types.NamespacedName{value.Namespace, value.Name}
            services[name] = value
        }
    default:
        glog.V(4).Infof("Received invalid update
type: %v", update)
    }

```



```
s.services[source] = services
s.serviceLock.Unlock()
if s.updates != nil {
    s.updates <- struct{}{}
}
return nil
}
```

serviceStore同时是config.Accessor接口的一个实现，MergedState接口方法返回之前Merge最新的Service全量数据。

```
func (s *serviceStore) MergedState() interface{} {
    s.serviceLock.RLock()
    defer s.serviceLock.RUnlock()
    services := make([]api.Service, 0)
    for _, sourceServices := range s.services {
        for _, value := range sourceServices {
            services = append(services, value)
        }
    }
    return services
}
```

上述方法在哪里被用到了呢？就在之前提到的NewServiceConfig方法里：

```
go watchForUpdates(bcaster, store, updates)
```

一个协程监听serviceStore的updates（Channel），在收到事件以后就调用上述MergedState方法，将当前最新的Service数组通知注册到bcaster上的所有Listener进行处理。下面分别给出了watchForUpdates及Broadcaster的Notify方法的源码：

```
func watchForUpdates(bcaster *config.Broadcaster,
accessor config.Accessor, updates <-chan struct{}) {
    for true {
        <-updates
        bcaster.Notify(accessor.MergedState())
    }
}

func (b *Broadcaster) Notify(instance interface{}) {
    b.listenerLock.RLock()
    listeners := b.listeners
    b.listenerLock.RUnlock()
    for _, listener := range listeners {
        listener.OnUpdate(instance)
    }
}
```

上述逻辑的精巧设计之处在于，当ServiceConfig完成Merge调用后，为了及时通知Listener进行处理，就产生一个“空事件”并写入updates这个Channel中，另外监听此Channel的协程就及时得到通知，触发Listener的回调动作。ServiceConfig这里注册的Listener是proxy.Proxier对象，我们以后会继续分析它的回调函数OnUpdate是如何使用Service数据的。

接下来，我们看看 ServiceUpdate 事件是怎么生成并传递到 ServiceConfig 的 Channel 上的。在 kube-proxy 启动流程中有调用 config.NewSourceAPI 函数，其内部生成了一个 servicesReflector 对象：

```
type servicesReflector struct {  
    watcher          ServicesWatcher  
    services          chan<- ServiceUpdate  
    resourceVersion   string  
    waitDuration      time.Duration  
    reconnectDuration time.Duration  
}
```

其中 services 这个 Channel 是用来写入 ServiceUpdate 事件的，它是 ServiceConfig 的 Channel（source string）方法所创建并返回的 Channel，它写入数据后就会被一个协程立即转发到 ServiceConfig 的 Channel 里。下面这段代码完整地揭示了上述逻辑：

```
func (c *ServiceConfig) Channel(source string) chan  
ServiceUpdate {  
    ch := c.mux.Channel(source)  
    serviceCh := make(chan ServiceUpdate)  
    go func() {  
        for update := range serviceCh {  
            ch <- update  
        }  
        close(ch)  
    }()  
}
```

```
        return serviceCh
    }
}
```

servicesReflector 中的 watcher 用来从 API Server 上拉取 Service 数据，它是 client.Services (api.NamespaceAll) 返回的 client.ServiceInterface 实例对象的一个引用，属于标准的 Kubernetes client 包。在 config.NewSourceAPI 的方法里，启动了一个协程周期性地调用 watcher 的 list 与 Watch 方法获取数据，然后转换成 ServiceUpdate 事件，写入 Channel 中。下面是关键源码：

```
func (s *servicesReflector) run(resourceVersion
*string) {
    if len(*resourceVersion) == 0 {
        services, err :=
s.watcher.List(labels.Everything())
        if err != nil {
            glog.Errorf("Unable to load
services: %v", err)

            // TODO: reconcile with
pkg/client/cache which doesn't use reflector.

            time.Sleep(wait.Jitter(s.waitDuration, 0.0))

            return
        }
        *resourceVersion = services.ResourceVersion
        // TODO: replace with code to update the
        s.services <- ServiceUpdate{Op: SET,
```

```

Services: services.Items}
    }

    watcher, err :=
s.watcher.Watch(labels.Everything(), fields.Everything(),
*resourceVersion)
    if err != nil {
        glog.Errorf("Unable to watch for services
changes: %v", err)
        if !client.IsTimeout(err) {
            // Reset so that we do a fresh get
request
            *resourceVersion = ""
        }
        time.Sleep(wait.Jitter(s.waitDuration,
0.0))
        return
    }
    defer watcher.Stop()
    ch := watcher.ResultChan()
    s.watchHandler(resourceVersion, ch, s.services)
}

```

在上面的代码中，初始时资源版本变量`resourceVersion`为空，于是会执行**Service**的全量拉取动作（`watcher.List`），之后**Watch**资源会开始发生变化（`watcher.Watch`）并将**Watch**的结果（一个**Channel**保持了**Service**的变动数据）也转换为对应的**ServiceUpdate**事件并写入**Channel**中。另外，当拉取数据的调用发生异常时，`resourceVersion`恢复为空，

导致重新进行全量资源的拉取动作。这种自修复能力的编程设计足以见证谷歌大神们的深厚编程功力；另外，笔者认为kubernetes这里的ServiceConfig的设计实现思路和代码要比kubenet中的好一点，虽然两个作者都是顶尖高手。

接下来才开始进入本节的重点，即服务代理的实现机制分析。首先，我们从代码中的load balance组件说起。下面是kubernetes中定义的LoadBalancer接口：

```
type LoadBalancer interface {
    NextEndpoint(service ServicePortName, srcAddr
net.Addr) (string, error)
    NewService(service ServicePortName,
sessionAffinityType          api.ServiceAffinity,
stickyMaxAgeMinutes int) error
    CleanupStaleStickySessions(service ServicePortName)
}
```

LoadBalancer有3个接口，其中NextEndpoint方法用于给访问指定Service的新客户端请求分配一个可用的Endpoint地址；NewService用来添加一个新服务到负载均衡器上；CleanupStaleStickySessions则用来清理过期的Session会话。目前kubernetes只实现了一个基于round-robin算法的负载均衡器，它就是proxy.LoadBalancerRR组件。

LoadBalancerRR采用了affinityState这个结构体来保存当前客户端的会话信息，然后在affinityPolicy里用一个Map来记录（属于某个Service的）所有活动的客户端会话，这是它实现Session亲和性的负载均衡调度的基础。

```
type affinityState struct {
    clientIP string
    //clientProtocol api.Protocol //not yet used
    //sessionCookie string //not yet used
    endpoint string
    lastUsed time.Time
}

type affinityPolicy struct {
    affinityType api.ServiceAffinity
    affinityMap map[string]*affinityState // map
client IP -> affinity info
    ttlMinutes int
}
```

balancerState用来记录一个**Service**的所有**Endpoint**（数组）、当前所使用的**Endpoint**的**index**，以及对应的所有活动的客户端会话（**affinityPolicy**）。其定义如下：

```
type balancerState struct {
    endpoints []string // a list of "ip:port" style
strings
    index int // current index into endpoints
    affinity affinityPolicy
}
```

有了上面的认识，再看**LoadBalancerRR**的构造函数就简单多了，它内部用一个**map**记录每个服务的**balancerState**状态，当然初始化时还

是空的:

```
func NewLoadBalancerRR() *LoadBalancerRR {  
    return &LoadBalancerRR{  
  
                                services:  
map[ServicePortName]*balancerState{},  
    }  
}
```

LoadBalancerRR 的 **NewService** 方法代码很简单，就是在它的 **services** 里增加一个记录项，用户端的会话超时时间 **ttlMinutes** 默认为3小时，下面是相关源码:

```
func (lb *LoadBalancerRR) NewService(svcPort  
ServicePortName, affinityType api.ServiceAffinity,  
ttlMinutes int) error {  
    lb.lock.Lock()  
    defer lb.lock.Unlock()  
    lb.newServiceInternal(svcPort, affinityType,  
ttlMinutes)  
    return nil  
}  
  
func (lb *LoadBalancerRR) newServiceInternal(svcPort  
ServicePortName, affinityType api.ServiceAffinity,  
ttlMinutes int) *balancerState {  
    if ttlMinutes == 0 {  
        ttlMinutes = 180
```



```

    }
    if _, exists := lb.services[svcPort]; !exists {
        lb.services[svcPort] =
&balancerState{affinity:    *newAffinityPolicy(affinityType,
ttlMinutes)}

        glog.V(4).Infof("LoadBalancerRR service %q
did not exist, created", svcPort)
    } else if affinityType != "" {
        lb.services[svcPort].affinity.affinityType
= affinityType
    }
    return lb.services[svcPort]
}

```

我们在前面提到过 ServiceConfig 同步并监听 API Server 上的 api.Service 的数据变化，然后调用 Listener（proxy.Proxyer 是 ServiceConfig 唯一注册的 Listener）的 OnUpdate 接口完成通知。而上述 NewService 就是在 proxy.Proxyer 的 OnUpdate 方法里被调用的，从而实现了 Service 自动添加到 LoadBalancer 的机制。

我们再来看 LoadBalancerRR 的 NextEndpoint 方法，它实现了经典的 round-robin 负载均衡算法。NextEndpoint 方法首先判断当前服务是否有保持会话（sessionAffinity）的要求，如果有，则看当前请求是否有连接可用：

```

if sessionAffinityEnabled {
    // Caution: don't shadow ipaddr

```

```

        var err error

                                ipaddr, _, err =
net.SplitHostPort(srcAddr.String())
        if err != nil {
            return "", fmt.Errorf("malformed
source address %q: %v", srcAddr.String(), err)
        }

                                sessionAffinity, exists :=
state.affinity.affinityMap[ipaddr]

                                if exists &&
int(time.Now().Sub(sessionAffinity.lastUsed).Minutes()) <
state.affinity.ttlMinutes {
                                // Affinity wins.

                                endpoint :=
sessionAffinity.endpoint

                                sessionAffinity.lastUsed =
time.Now()

                                glog.V(4).Infof("NextEndpoint for
service %q from IP %s with sessionAffinity %+v: %s",
svcPort, ipaddr, sessionAffinity, endpoint)
                                return endpoint, nil
        }
    }
}

```

如果服务无须会话保持、新建会话及会话过期，则采用round-robin算法得到下一个可用的服务端口，如果服务有会话保持需求，则保存当前的会话状态：

```
// Take the next endpoint.
    endpoint := state.endpoints[state.index]
        state.index = (state.index + 1) %
len(state.endpoints)
    if sessionAffinityEnabled {
        var affinity *affinityState
                                affinity =
state.affinity.affinityMap[ipaddr]
        if affinity == nil {
            affinity = new(affinityState)
//&affinityState{ipaddr, "TCP", "", endpoint, time.Now()}
            state.affinity.affinityMap[ipaddr]
= affinity
        }
        affinity.lastUsed = time.Now()
        affinity.endpoint = endpoint
        affinity.clientIP = ipaddr
        glog.V(4).Infof("Updated affinity key %s:
%+v", ipaddr, state.affinity.affinityMap[ipaddr])
    }
    return endpoint, nil
```

接下来我们看看 Service 的 Endpoint 信息是如何添加到 LoadBalancerRR 上的？答案很简单，类似之前我们分析过的 ServiceConfig。kube-proxy 也设计了一个 EndpointsConfig 来拉取和监听 API Server 上的服务的 Endpoint 信息，并调用 LoadBalancerRR 的

OnUpdate接口完成通知，在这个方法里，LoadBalancerRR完成了服务访问端口的添加和同步逻辑。

我们先来看看api.Endpoints的定义：

```
type EndpointAddress struct {
    IP string
    TargetRef *ObjectReference
}
type EndpointPort struct {
    Name string
    Port int
    Protocol Protocol
}
type EndpointSubset struct {
    Addresses []EndpointAddress
    Ports      []EndpointPort
}
type Endpoints struct {
    TypeMeta    `json:",inline"`
    ObjectMeta  `json:"metadata,omitempty"`
    Subsets []EndpointSubset
}
```

一个EndpointAddress与EndpointPort对象可以组成一个服务访问地址，而在EndpointSubset对象里则定义了两个单独的EndpointAddress与EndpointPort数组而不是“服务访问地址”的一个列表。初看这样的定义

你可能会觉得很奇怪，为什么没有设计一个Endpoint结构？这里的深层次原因在于，Service的Endpoint信息来源于两个独立的实体：Pod与Service，前者负责提供IP地址即EndpointAddress，而后者负责提供Port即EndpointPort。由于在一个Pod上可以运行多个Service，而一个Service也通常跨越多个Pod，于是就产生了一个“笛卡尔乘积”的Endpoint列表，这就是EndpointSubset的设计灵感。

举例说明，对于如下表示的EndpointSubset:

```
{
    Addresses: [{"ip": "10.10.1.1"}, {"ip":
"10.10.2.2"}],
    Ports: [{"name": "a", "port": 8675}, {"name":
"b", "port": 309}]
}
```

会产生如下Endpoint列表:

```
a: [ 10.10.1.1:8675, 10.10.2.2:8675 ],
b: [ 10.10.1.1:309, 10.10.2.2:309 ]
```

LoadBalancerRR的OnUpdate方法里循环对每个api.Endpoints进行处理，先把它转化为一个Map，Map的Key是EndpointPort的Name属性（代表一个Service的访问端口）；而Value则是hostPortPair的一个数组，hostPortPair其实就是之前缺失的Endpoint结构体，包括一个IP地址与端口属性，即某个服务在一个Pod上的对应访问端口。

```

portsToEndpoints := map[string][]hostPortPair{
    for i := range svcEndpoints.Subsets {
        ss := &svcEndpoints.Subsets[i]
        for i := range ss.Ports {
            port := &ss.Ports[i]
            for i := range ss.Addresses
{
                                                                    addr :=
&ss.Addresses[i]

portsToEndpoints[port.Name] =
append(portsToEndpoints[port.Name],    hostPortPair{addr.IP,
port.Port})

                                                                    // Ignore the
protocol field - we'll get that from the Service objects.
            }
        }
    }
}

```

下一步，针对`portsToEndpoints`进行循环处理。对于每个记录，判断是否已经在`services`中存在，并做出相应的更新或跳过的逻辑，最后删除那些已经不在集合中的端口，完成整个同步逻辑。下面是相关代码：

```

for portname := range portsToEndpoints {
                                                                    svcPort :=
ServicePortName{types.NamespacedName{svcEndpoints.Namespace

```


is called.

```

                                                                    state =
lb.newServiceInternal(svcPort, api.ServiceAffinity(""), 0)
                                                                    state.endpoints =
slice.ShuffleStrings(newEndpoints)

                                                                    // Reset the round-robin
index.

                                                                    state.index = 0
                                                                    }
                                                                    registeredEndpoints[svcPort] = true
                                                                    }
                                                                    }
                                                                    // Remove endpoints missing from the update.
                                                                    for k := range lb.services {
                                                                    if _, exists := registeredEndpoints[k];
!exists {
                                                                    glog.V(2).Infof("LoadBalancerRR:
Removing endpoints for %s", k)
                                                                    delete(lb.services, k)
                                                                    }
                                                                    }
                                                                    }
```

LoadBalancerRR的代码总体来说还是比较简单的，它主要被kubernetes中的关键组件proxy.Proxyer所使用，后者用到的主要数据结构为proxy.serviceInfo，它定义和保存了一个Service的代理过程中的必要参数和对象。下面是其定义：

```
type serviceInfo struct {  
    portal                portal  
    protocol              api.Protocol  
    proxyPort             int  
    socket                proxySocket  
    timeout               time.Duration  
    nodePort              int  
    loadBalancerStatus    api.LoadBalancerStatus  
    sessionAffinityType    api.ServiceAffinity  
    stickyMaxAgeMinutes   int  
    // Deprecated, but required for back-compat  
    (including e2e)  
    deprecatedPublicIPs []string  
}
```

serviceInfo的各个属性解释如下。

- portal: 用于存放服务的Portal地址，即Service的ClusterIP（VIP）地址与端口。
- protocol: 服务的TCP，目前是TCP与UDP。
- socket、proxyPort: socket是Proxier在本机上为该服务打开的代理Socket；proxyPort则是这个代理Socket的监听端口。
- timeout: 目前只用于UDP的Service，表明服务“链接”的超时时间。
- nodePort: 该服务定义的NodePort。
- loadBalancerStatus: 在Cloud环境下，如果存在由Cloud服务提供者提供的负载均衡器（软件或硬件）用作Kubernetes Service的负

载均衡，则这里存放这些负载均衡器的IP地址。

- `sessionAffinityType`: 该服务的负载均衡调度是否保持会话。
- `stickyMaxAgeMinutes`: 即前面说的Session过期时间。
- `deprecatedPublicIPs`: 已过期、废弃的服务的Public IP地址。

理解了`serviceInfo`，我们再来看Proxier的数据结构：

```
type Proxier struct {  
    loadBalancer LoadBalancer  
    mu            sync.Mutex // protects serviceMap  
    serviceMap    map[ServicePortName]*serviceInfo  
    portMapMutex  sync.Mutex  
    portMap       map[portMapKey]ServicePortName  
    numProxyLoops int32  
    listenIP      net.IP  
    iptables      iptables.Interface  
    hostIP        net.IP  
    proxyPorts    PortAllocator  
}
```

Proxier用一个Map维护了每个服务的`serviceInfo`信息，同时为了快速查询和检测服务端口是否有冲突，比如定义了两个一样端口的服务，又设计了一个`portMap`，其Key为服务的端口信息（`portMapKey`由port和protocol组合而成），value为`ServicePortName`。Proxier的`listenIP`为Proxier监听的本节点IP，它在这个IP上接收请求并做转发代理。由于每个服务的`proxySocket`在本节点监听的Port端口默认是系统随机分配的，所以使用`PortAllocator`来分配这个端口。另外，Service的Portal

与NodePort是通过Linux防火墙机制来实现的，因此这里引用了Iptables的组件完成相关操作。

要想理解 Proxier 中使用 Iptables 的方式，首先我们要弄明白 Kubernetes 中 Service 访问的一些网络细节。先来看看图6.9，这是一个外部应用通过NodePort（TCP: //NodeIP: NodePort）来访问Service时的网络流量示意图。访问流量进入节点网卡eth0后，到达Iptables的PREROUTING链，通过KUBE-NODEPORT-CONTAINER这个NAT规则被转发到kube-proxy进程上该Service对应的Proxy端口，然后由kube-proxy进程进行负载均衡并且将流量转发到Service所在Container的本地端口。

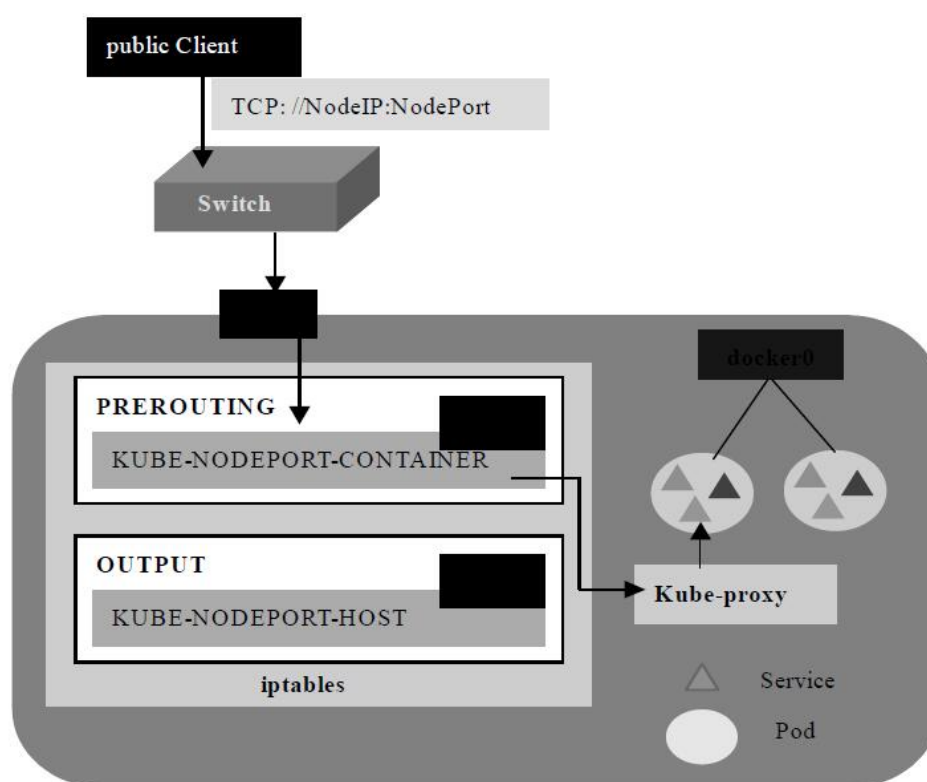


图6.9 外部应用通过NodePort访问Service的网络流量示意图

根据 Iptables 的机制，本地进程发起的流量会经过 Iptables 的 OUTPUT 链，于是 kube-proxy 在这里也增加了相同作用的 NAT 规则：KUBE-NODEPORT-HOST。这样一来，如果本地容器内的进程以 NodePort 方式来访问 Service，则流量也会被转发到 kube-proxy 上，虽然以这种方式访问的情况比较少见。

服务之间通过 Service Portal 方式访问的流量转发机制跟 NodePort 方式在本质上是一样的，也是通过 NAT，如图 6.10 所示。当 Service A 用 Service B 的 Portal 地址去访问时，流量经过 Iptables 的 OUTPUT 链经 NAT 规则 KUBE-PORTALS-HOST 的转换被转发到 kube-proxy 上，然后被转发给 Service B 所在的容器。

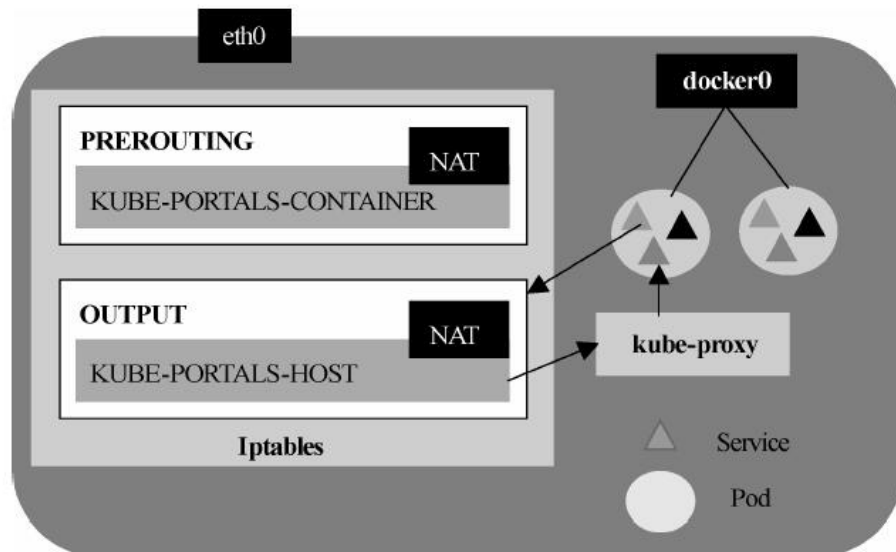


图6.10 以Service Portal方式访问Service的流量示意图

Proxier在创建Iptables的PREROUTING链中的NAT转发规则时，有一些特殊性，源码作者在代码中做了如下注释：

“这是一个复杂的问题。

如果Proxy的Proxier.listenIP设置为0.0.0.0，即绑定到所有端口上，那么我们采用REDIRECT这种方式进行流量转发，因为这种情况下，返回的流量与进入的流量使用同一个网络端口，这就满足了NAT的规则。其他情况则采用DNAT转发流量，但DNAT到127.0.0.1时，流量会消失，这似乎是Iptables的一个众所周知的问题，所以这里不允许Proxy绑定到localhost上。”

现在再看下面这段代码就容易理解了，用来生成KUBE-NODEPORT-CONTAINER这条NAT规则：

```
func (proxier *Proxier)
iptablesContainerNodePortArgs(nodePort int, protocol
api.Protocol, proxyIP net.IP, proxyPort int, service
ServicePortName) []string {
    args := iptablesCommonPortalArgs(nil, nodePort,
protocol, service)
    if proxyIP.Equal(zeroIPv4) ||
proxyIP.Equal(zeroIPv6) {
        // TODO: Can we REDIRECT with IPv6
        args = append(args, "-j", "REDIRECT", "--
to-ports", fmt.Sprintf("%d", proxyPort))
    } else {
        // TODO: Can we DNAT with IPv6
```

```

        args = append(args, "-j", "DNAT", "--to-destination",
                        net.JoinHostPort(proxyIP.String(),
                        strconv.Itoa(proxyPort)))
    }
    return args
}

```

弄明白Proxier中关于Iptables的事情之后，我们来研究分析下Proxier如何在OnUpdate方法里为每个Service建立起对应的Proxy并完成同步工作。首先，在OnUpdate方法里创建一个map（activeServices）来标识当前所有alive的Service，key为ServicePortName，然后对OnUpdate参数里的Service数组进行循环，判断每个Service是否需要新建、变更或者删除操作，对于需要新建或者变更的Service，先用PortAllocator获取一个新的未用的本地代理端口，然后调用addServiceOnPort方法创建一个ProxySocket用于实现此服务的代理，接着调用openPortal方法添加iptables里的NAT映射规则，最后调用LoadBalancer的NewService方法把该服务添加到负载均衡器上。OnUpdate方法的最后一段逻辑是处理已经被删除的Service，对于每个要被删除的Service，先删除Iptables中相关的NAT规则，然后关闭对应的proxySocket，最后释放ProxySocket占用的监听端口并将该端口“还给”PortAllocator。

从上面的分析中，我们看到addServiceOnPort是Proxier的核心方法之一。下面是该方法的源码：

```

func (proxier *Proxier) addServiceOnPort(service
ServicePortName, protocol api.Protocol, proxyPort int,

```

```

timeout time.Duration) (*serviceInfo, error) {
    sock, err := newProxySocket(protocol,
proxier.listenIP, proxyPort)
    if err != nil {
        return nil, err
    }

    _, portStr, err :=
net.SplitHostPort(sock.Addr().String())
    if err != nil {
        sock.Close()
        return nil, err
    }
    portNum, err := strconv.Atoi(portStr)
    if err != nil {
        sock.Close()
        return nil, err
    }
    si := &serviceInfo{
        proxyPort:      portNum,
        protocol:        protocol,
        socket:          sock,
        timeout:         timeout,
        sessionAffinityType: api.ServiceAffinityNone,
// default
                                stickyMaxAgeMinutes: 180,
// TODO: paramaterize this in the API.
    }

```

```
proxier.setServiceInfo(service, si)

    glog.V(2).Infof("Proxying for service %q on %s
port %d", service, protocol, portNum)
    go func(service ServicePortName, proxier *Proxier)
    {
        defer util.HandleCrash()
        atomic.AddInt32(&proxier.numProxyLoops, 1)
        sock.ProxyLoop(service, si, proxier)
        atomic.AddInt32(&proxier.numProxyLoops, -1)
    }(service, proxier)

    return si, nil
}
```

在上述代码中，先创建一个 **ProxySocket**，然后创建一个 **serviceInfo** 并添加到 **Proxier** 的 **serviceMap** 中，最后启动一个协程调用 **ProxySocket** 的 **ProxyLoop** 方法，使得 **ProxySocket** 进入 **Listen** 状态，开始接收并转发客户端请求。

kube-proxy 中的 **ProxySocket** 有两个实现，其中一个 是 **tcpProxySocket**，另外一个 是 **udpProxySocket**，二者的工作原理都一样，它们的工作流程就是为每个客户端 **Socket** 请求创建一个到 **Service** 的后端 **Socket** 连接，并且“打通”这两个 **Socket**，即把客户端 **Socket** 发来的数据“复制”到对应的后端 **Socket** 上，然后把后端 **Socket** 上服务响应的数据写入客户端 **Socket** 上去。

以tcpProxySocket为例，我们先看看它是如何完成Service后端连接创建过程的：

```
func tryConnect(service ServicePortName, srcAddr
net.Addr, protocol string, proxier *Proxier) (out net.Conn,
err error) {
    for _, retryTimeout := range endpointDialTimeout {
        endpoint, err :=
proxier.loadBalancer.NextEndpoint(service, srcAddr)
        if err != nil {
            glog.Errorf("Couldn't find an
endpoint for %s: %v", service, err)
            return nil, err
        }
        glog.V(3).Infof("Mapped service %q to
endpoint %s", service, endpoint)
        outConn, err := net.DialTimeout(protocol,
endpoint, retryTimeout*time.Second)
        if err != nil {
            if isTooManyFDsError(err) {
                panic("Dial failed: " +
err.Error())
            }
            glog.Errorf("Dial failed: %v", err)
            continue
        }
        return outConn, nil
    }
}
```

```
    }  
    return nil, fmt.Errorf("failed to connect to an  
endpoint.")  
}
```

在上述方法里，首先调用`loadBalancer.NextEndpoint`方法获取服务的下一个可用`Endpoint`地址，然后调用标准网络库中的方法建立到此地址的连接，如果连接失败，则会重新尝试，间隔时间指数增加（参见`endpointDialTimeout`的值）。

在后端`Service`的连接建立以后，`proxyTCP`方法就会启动两个协程，通过调用Go标准库`io`里的`Copy`方法把输入流的数据写入输出流，从而完成前后端连接的数据转发功能。此外，`proxyTCP`方法会阻塞，直到前后端两个连接的数据流都关闭（或结束）才会返回。下面是其源码：

```
func proxyTCP(in, out *net.TCPConn) {  
    var wg sync.WaitGroup  
    wg.Add(2)  
    glog.V(4).Infof("Creating proxy between %v <-> %v  
<-> %v <-> %v",  
                    in.RemoteAddr(), in.LocalAddr(),  
out.LocalAddr(), out.RemoteAddr())  
    go copyBytes("from backend", in, out, &wg)  
    go copyBytes("to backend", out, in, &wg)  
    wg.Wait()  
    in.Close()
```

```
        out.Close()  
    }
```

这里我们留一个问题，**kube-proxy**会在当前节点上为每个**Service**都建立一个代理么？不管本节点上是否有该**Service**对应的**Pod**？

6.6.3 设计总结

从之前的启动流程和代码分析来看，`kube-proxy`的设计和实现还是比较精巧和紧凑的，它的流程只有一个：从Kubernetes API Server上同步Service及其Endpoint信息，为每个Service建立一个本地代理以完成具备负载均衡能力的服务转发功能。图6.11给出了`kube-proxy`的总体设计示意图，为了清晰地表明整个业务流程和数据传递方向，这里省去了一些非关键的结构体和对象。`app.ProxyServer`创建了一个`config.SourceAPI`的结构体，用于拉取Kubernetes API Server上的Service与Endpoint配置信息，分别由`config.servicesReflector`与`config.endpointsReflector`这两个对象来实现，它们各自通过相应的Kubernetes Client API来拉取数据并且生成对应的Update信息放入Channel中，最终Channel中的Service数据到达`proxy.Proxier`上，`proxy.Proxier`为每个Service建立一个`proxySocket`实现服务代理并且在iptables上创建相关的NAT规则，然后在LoadBalancer组件上开通该服务的负载均衡功能；而Channel中的Endpoints数据则被发送到`proxy.LoadBalancerRR`组件，用于给每个服务建立一个负载均衡的状态机，每个服务用`balancerState`结构体来保存该服务可用的Endpoint地址及当前的会话状态`affinityPolicy`，对于需要保存会话状态的服务，`affinityPolicy`用一个Map来存储每个客户的会话状态`affinityState`。

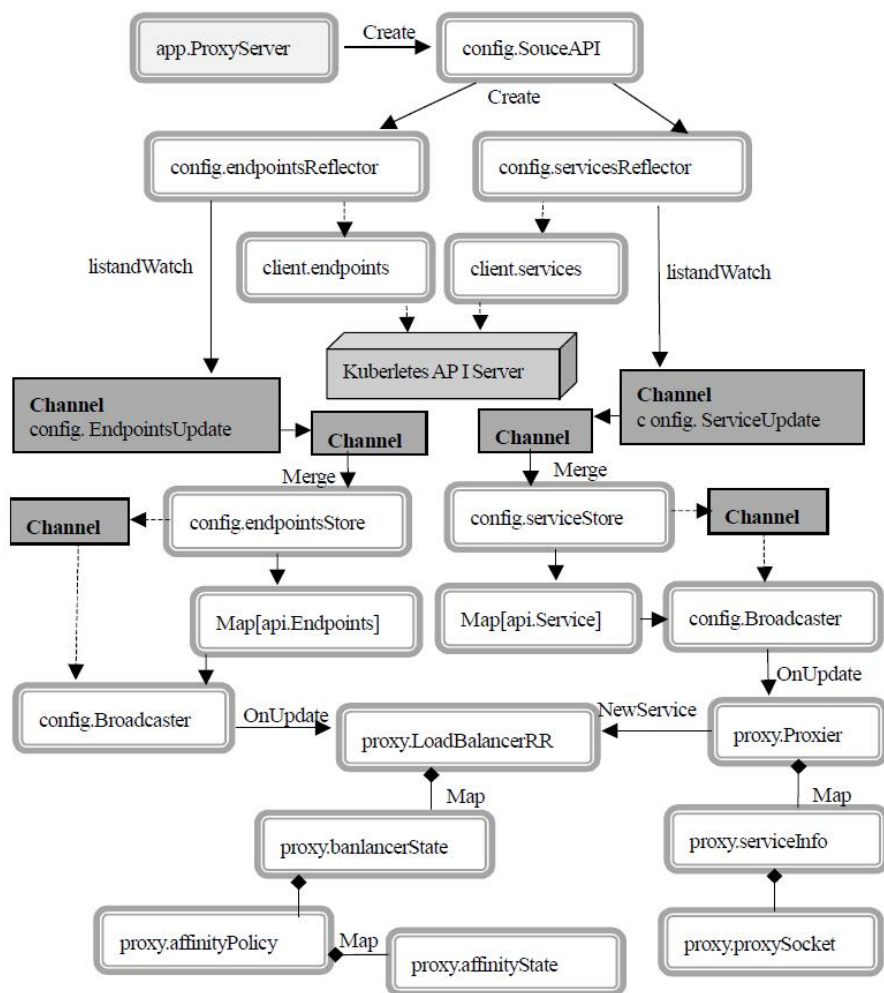


图6.11 与kubelet总体相关的设计示意图

6.7 kubectl进程源码分析

kubectl与之前的Kubernetes进程不同，它不是一个后台运行的守护进程，而是Kubernetes提供的一个命令行工具（CLI），它提供了一组命令来操作Kubernetes集群。

kubectl进程的入口类源码位置如下：

```
github.com/GoogleCloudPlatform/kubernetes/cmd/kubectl/kubec  
tl.go
```

入口main（）函数的逻辑很简单：

```
func main() {  
    runtime.GOMAXPROCS(runtime.NumCPU())  
    cmd := cmd.NewKubectlCommand(cmdutil.NewFactory(nil),  
os.Stdin, os.Stdout, os.Stderr)  
    if err := cmd.Execute(); err != nil {  
        os.Exit(1)  
    }  
}
```

上述代码通过NewKubectlCommand方法创建了一个具体的Command命令并调用它的Execute方法执行，这是工厂模式结合命令模

式的一个经典设计案例。从NewKubectlCommand的源码中可以看到，kubectl 的 CLI 命令 框 架 使 用 了 GitHub 开 源 项 目（<https://github.com/spf13/cobra>），下面是该框架中对Command的定义：

```
type Command struct {  
    Use string // The one-line usage message.  
    Short string // The short description shown in the  
'help' output.  
    Long string // The long message shown in the 'help  
<this-command>' output.  
    Run func(cmd *Command, args []string) // Run runs  
the command.  
}
```

实现一个具体Command就只要实现Command的Run函数即可，下面是其官方网页给出的一个Echo命令的例子：

```
var cmdEcho = &cobra.Command{  
    Use: "echo [string to echo]",  
    Short: "Echo anything to the screen",  
    Long: `echo is for echoing anything back.  
    Echo works a lot like print, except it has a  
child command.  
`,  
    Run: func(cmd *cobra.Command, args []string) {  
        fmt.Println("Print: " + strings.Join(args,
```

```
    ""))  
    },  
}
```

由于大多数kubectI的命令都需要访问Kubernetes API Server，所以kubectI设计了一个类似命令的上下文环境的对象——util.Factory供Command对象使用。

在接下来的几个章节中，我们对kubectI中的几个典型Command的源码逐一解读。

6.7.1 kubectl create命令

`kubectl create`命令通过调用Kubernetes API Server提供的RestAPI来创建Kubernetes资源对象，例如Pod、Service、RC等，资源的描述信息来自-f指定的文件或者来自命令行的输入流。下面是创建create命令的相关源码：

```
func NewCmdCreate(f *cmdutil.Factory, out io.Writer)
*cobra.Command {
    var filenames util.StringList
    cmd := &cobra.Command{
        Use:      "create -f FILENAME",
        Short:    "Create a resource by filename or
stdin",
        Long:     create_long,
        Example:  create_example,
        Run: func(cmd *cobra.Command, args
[]string) {
            cmdutil.CheckErr(ValidateArgs(cmd,
args))
            cmdutil.CheckErr(RunCreate(f, out,
filenames))
        },
    }
}
```

```
        usage := "Filename, directory, or URL to file to  
use to create the resource"  
  
        kubect1.AddJsonFilenameFlag(cmd, &filenames, usage)  
        cmd.MarkFlagRequired("filename")  
        return cmd  
    }
```

`AddJsonFilenameFlag`方法限制filename参数（-f）的文件名后缀只能是json、yaml或者yml中的一种，并且将参数值填充到filenames这个Set集合中，随后被Command的Run函数中的RunCreate方法所引用，后者就是kubect1 create命令的核心逻辑所在。

`RunCreate`方法使用到了resource.Builder对象，它是kubect1中的一处复杂设计，采用了Visitor的设计模式，kubect1的很多命令都用到了它。Builder的目标是根据命令行输入的资源相关的参数，创建针对性的Visitor对象来获取对应的资源，最后遍历相关的所有Visitor对象，触发用户指定的VisitorFun回调函数来处理每个具体的资源，最终完成资源对象的业务处理逻辑。由于涉及的资源参数有各种情况，所以导致Builder的代码很复杂。以下是Builder所能操作的各种资源参数：

- 通过输入流提供具体的资源描述；
- 通过本地文件内容或者HTTP URL的输出流来获取资源描述；
- 文件列表提供多个资源描述；
- 指定资源类型，通过查询Kubernetes API Server来获取相关类型的资源；
- 指定资源的selector条件如cluster-service=true，查询Kubernetes API Server来获取相关的资源；

- 指定资源的namespace来查询符合条件的相关资源。

下面是resource.Builder的定义:

```
type Builder struct {  
    mapper *Mapper  
    errs []error  
    paths []Visitor  
    stream bool  
    dir    bool  
    selector labels.Selector  
    selectAll bool  
    resources []string  
    namespace string  
    names      []string  
    resourceTuples []resourceTuple  
    defaultNamespace bool  
    requireNamespace bool  
    flatten bool  
    latest bool  
    requireObject bool  
    singleResourceType bool  
    continueOnError bool  
    schema validation.Schema  
}
```

其实Builder很像一个SQL查询条件的生成器，里面包括了各种“查询”条件，在指定不同的查询条件时，会生成不同的Visitor接口来处理这些查询条件，最后遍历所有Visitor，就得到最终的“查询结果”。Builder返回的Result对象里也包括Visitor对象及可能的最终资源列表等信息，由于资源查询存在各种情况，所以Result也提供了多种方法，比如还包括了Watch资源变化的方法。

RunCreate方法里先创建了一个Builder，设置各种必要参数，然后调用Builder的Do方法，返回一个Result，代码如下：

```
    schema, err := f.Validator()
    mapper, typer := f.Object()
        r := resource.NewBuilder(mapper, typer,
f.ClientMapperForCommand()).
        Schema(schema).
        ContinueOnError().

NamespaceParam(cmdNamespace).DefaultNamespace().
        FilenameParam(enforceNamespace,
filenames...).
        Flatten().
        Do()
```

其中，schema对象用来校验资源描述是否正确，比如有没有缺少字段或者属性的类型错误等；mapper对象用来完成从资源描述信息到资源对象的转换，用来在REST调用过程中完成数据转换；FilenameParam是这里唯一指定Builder的资源参数的方法，即把命令行

传入的**filenames**参数作为资源参数；**Flatten**方法则告诉**Builder**，这里的资源对象其实是一个数组，需要**Builder**构造一个**FlattenListVisitor**来遍历**Visit**数组中的每个资源项目；**Do**方法则返回一个**Rest**对象，里面包括与资源相关的**Visitor**对象。

下面是**NamespaceParam**方法的源码，主要逻辑为调用**Builder**的**Builder.Stdin**、**Builder.URL**或**Builder.Path**方法来处理不同类型的资源参数，这些方法会生成对应的**Visitor**对象并加入**Builder**的**Visitor**数组里（**paths**属性）。

```
func (b *Builder) FilenameParam(enforceNamespace bool,
paths ...string) *Builder {
    for _, s := range paths {
        switch {
        case s == "-":
            b.Stdin()
        case strings.Index(s, "http://") == 0 ||
strings.Index(s, "https://") == 0:
            url, err := url.Parse(s)
            if err != nil {
                b.errs = append(b.errs, fmt.Errorf("the URL passed to
filename %q is not valid: %v", s, err))
            }
            continue
        }
        b.URL(url)
    }
    default:
        b.Path(s)
    }
```

```

    }
    }
    if enforceNamespace {
        b.RequireNamespace()
    }
    return b
}

```

不管是标准输入流、URL，还是文件目录或者文件本身，这里处理资源的 Visitor 都是 StreamVisitor 这个实现（FileVisitor 与 FileVisitorForSTDIN 是 StreamVisitor 的一个 Wrapper）。下面是 StreamVisitor 的 Visit 接口代码：

```

func (v *StreamVisitor) Visit(fn VisitorFunc) error {
    d := yaml.NewYAMLorJSONDecoder(v.Reader, 4096)
    for {
        ext := runtime.RawExtension{}
        if err := d.Decode(&ext); err != nil {
            if err == io.EOF {
                return nil
            }
            return err
        }
        ext.RawJSON = bytes.TrimSpace(ext.RawJSON)
        if len(ext.RawJSON) == 0 ||
bytes.Equal(ext.RawJSON, []byte("null")) {
            continue
        }
    }
}

```

```

        }

        if err := ValidateSchema(ext.RawJSON,
v.Schema); err != nil {
            return err
        }

        info, err := v.InfoForData(ext.RawJSON,
v.Source)

        if err != nil {
            if v.IgnoreErrors {
                fmt.Fprintf(os.Stderr, "error: could
not read an encoded object from %s: %v\n", v.Source, err)
                glog.V(4).Infof("Unreadable: %s",
string(ext.RawJSON))
                continue
            }
            return err
        }

        if err := fn(info); err != nil {
            return err
        }
    }
}

```

在上述代码中，首先从输入流中解析具体的资源对象，然后创建一个**Info**结构体进行包装（转换后的资源对象存储在**Info**的**Object**属性中），最后再用这个**Info**对象作为参数调用回调函数**VisitorFunc**，从而完成整个逻辑流程。下面是**RunCreate**方法里调用**Builder**的**Visit**方法触

发Visitor执行时的源码，可以看到这里的VisitorFunc所做的事情是通过Rest Client发起Kubernetes API调用，把资源对象写入资源注册表里：

```
err = r.Visit(func(info *resource.Info) error {
                                                    data, err :=
info.Mapping.Codec.Encode(info.Object)
    if err != nil {
        return cmdutil.AddSourceToErr("creating",
info.Source, err)
    }
    obj, err := resource.NewHelper(info.Client,
info.Mapping).Create(info.Namespace, true, data)
    if err != nil {
        return cmdutil.AddSourceToErr("creating",
info.Source, err)
    }
    count++
    info.Refresh(obj, true)
    printObjectSpecificMessage(info.Object, out)
                                                    fmt.Fprintf(out, "%s/%s\n",
info.Mapping.Resource, info.Name)
    return nil
})
```

6.7.2 rolling-update命令

kubectl rolling-update 命令负责滚动更新（升级）RC（ReplicationController），下面是创建对应Command的源码：

```
func NewCmdRollingUpdate(f *cmdutil.Factory, out
io.Writer) *cobra.Command {
    cmd := &cobra.Command{
        Use: "rolling-update OLD_CONTROLLER_NAME
([NEW_CONTROLLER_NAME] -image=NEW_CONTAINER_IMAGE | -f
NEW_CONTROLLER_SPEC)",
        // rollingupdate is deprecated.
        Aliases: []string{"rollingupdate"},
        Short:   "Perform a rolling update of the
given ReplicationController.",
        Long:    rollingUpdate_long,
        Example: rollingUpdate_example,
        Run: func(cmd *cobra.Command, args
[]string) {
            err := RunRollingUpdate(f, out,
cmd, args)
            cmdutil.CheckErr(err)
        },
    }
}
```

```
cmd.Flags().String("update-period", updatePeriod,
`Time to wait between updating pods. Valid time units are
"ns", "us" (or "µs"), "ms", "s", "m", "h".`)
```

此处省去一些命令参数添加的非关键代码:

```
cmdutil.AddPrinterFlags(cmd)
return cmd
}
```

从上述代码中我们看到 **rolling-update** 命令的执行函数为 **RunRollingUpdate**，在分析这个函数之前，我们先了解下 **rolling-update** 执行过程中的一个关键逻辑。

rolling update动作可能由于网络超时或者用户等得不耐烦等原因被中断，因此我们可能会重复执行一条 **rolling-update** 命令，目的只有一个，就是恢复之前的 **rolling update** 动作。为了实现这个目的，**rolling-update** 程序在执行过程中会在当前 **rolling-update** 的 RC 上增加一个 **Annotation** 标签——**kubectrl.kubernetes.io/next-controller-id**，标签的值就是下一个要执行的新 RC 的名字。此外，对于 **Image** 升级这种更新方式，还会在 RC 的 **Selector** 上（**RC.Spec.Selector**）贴一个名为 **deploymentKey** 的 **Label**，**Label** 的值是 RC 的内容进行 **Hash** 计算后的值，相当于签名，这样就能很方便地比较 RC 里的 **Image** 名字（以及其他信息）是否发生了变化。

RunRollingUpdate 执行逻辑的第1步：确定 **New RC** 对象及建立起 **Old RC** 到 **New RC** 的关联关系。下面我们以指定的 **Image** 参数进行

rolling update的方式为例，看看代码是如何实现这段逻辑的。下面是相关源码：

```
    if len(image) != 0 {
        keepOldName = len(args) == 1
        newName := findNewName(args, oldRc)
                                if newRc, err =
kubect1.LoadExistingNextReplicationController(client,
cmdNamespace, newName); err != nil {
                                return err
        }
        if newRc != nil {
                                fmt.Fprintf(out, "Found existing
update in progress (%s), resuming.\n", newRc.Name)
        } else {
                                newRc, err =
kubect1.CreateNewControllerFromCurrentController(client,
cmdNamespace, oldName, newName, image, deploymentKey)
                                if err != nil {
                                    return err
                                }
        }

        // Update the existing replication
controller with pointers to the 'next' controller
        // and adding the <deploymentKey> label if
necessary to distinguish it from the 'next' controller.
        oldHash, err := api.HashObject(oldRc,
```

```

client.Codec)

        if err != nil {
            return err
        }

                                oldRc, err =
kubectl.UpdateExistingReplicationController(client, oldRc,
cmdNamespace, newRc.Name, deploymentKey, oldHash, out)
        if err != nil {
            return err
        }
    }
}

```

在代码里，findNewName方法查询新RC的名字，如果在命令行参数中没有提供新RC的名字，则从Old RC中根据kubectl.kubernetes.io/next-controller-id这个Annotation标签找新RC的名字并返回，如果新RC存在则继续使用，否则调用CreateNewControllerFromCurrentController方法创建一个新RC，在新RC的创建过程中设定deploymentKey的值为自己的Hash签名，方法源码如下：

```

func CreateNewControllerFromCurrentController(c
*client.Client, namespace, oldName, newName, image,
deploymentKey string) (*api.ReplicationController, error) {
    // load the old RC into the "new" RC

                                newRc, err :=
c.ReplicationControllers(namespace).Get(oldName)
    if err != nil {

```

```

        return nil, err
    }
    if len(newRc.Spec.Template.Spec.Containers) > 1 {
        // TODO: support multi-container image
update.
        return nil, goerrors.New("Image update is not
supported for multi-container pods")
    }
    if len(newRc.Spec.Template.Spec.Containers) == 0 {
        return nil, goerrors.New(fmt.Sprintf("Pod
has no containers! (%v)", newRc))
    }
    newRc.Spec.Template.Spec.Containers[0].Image =
image
    newHash, err := api.HashObject(newRc, c.Codec)
    if err != nil {
        return nil, err
    }
    if len(newName) == 0 {
        newName = fmt.Sprintf("%s-%s", newRc.Name,
newHash)
    }
    newRc.Name = newName
    newRc.Spec.Selector[deploymentKey] = newHash
    newRc.Spec.Template.Labels[deploymentKey] = newHash
    // Clear resource version after hashing so that
identical updates get different hashes.

```

```

        newRc.ResourceVersion = ""
        return newRc, nil
    }

```

在 Image rolling update 的流程中确定新的 RC 以后，调用 UpdateExistingReplicationController 方法，将旧 RC 的 kubectrl.kubernetes.io/next-controller-id 设置为新 RC 的名字，并且判断旧 RC 是否需要设置或更新 deploymentKey，具体代码如下：

```

func UpdateExistingReplicationController(c
client.Interface, oldRc *api.ReplicationController,
namespace, newName, deploymentKey, deploymentValue string,
out io.Writer) (*api.ReplicationController, error) {
    SetNextControllerAnnotation(oldRc, newName)
    if _, found := oldRc.Spec.Selector[deploymentKey];
!found {
                                                                    return
AddDeploymentKeyToReplicationController(oldRc, c,
deploymentKey, deploymentValue, namespace, out)
    } else {
        // If we didn't need to update the controller for
the deployment key, we still need to write
        // the "next" controller.
                                                                    return
c.ReplicationControllers(namespace).Update(oldRc)
    }
}

```

通过上面的逻辑，新RC被确定并且旧RC到新RC的关联关系也被建立好了，接下来如果dry-run参数为true，则仅仅打印新旧RC的信息然后返回。如果是正常的rolling update动作，则创建一个kubectl.RollingUpdater对象来执行具体任务，任务的参数则放在kubectl.RollingUpdaterConfig中，相关源码如下：

```
                                updateCleanupPolicy      :=
kubectl.DeleteRollingUpdateCleanupPolicy
                                if keepOldName {
                                                                    updateCleanupPolicy =
kubectl.RenameRollingUpdateCleanupPolicy
                                }
                                config := &kubectl.RollingUpdaterConfig{
                                    Out:          out,
                                    OldRc:       oldRc,
                                    NewRc:       newRc,
                                    UpdatePeriod: period,
                                    Interval:    interval,
                                    Timeout:     timeout,
                                    CleanupPolicy: updateCleanupPolicy,
                                }
```

其中out是输出流（屏幕输出）；UpdatePeriod是执行rolling update动作的间隔时间；Interval与Timeout组合使用，前者是每次拉取polling controller状态的间隔时间，而后者则是对应的（HTTP REST调用）超时时间。CleanupPolicy确定升级结束后的善后策略，比如DeleteRollingUpdateCleanupPolicy表示删除旧的RC，而

RenameRollingUpdateCleanupPolicy则表示保持RC的名字不变（改变新RC的名字）。

RollingUpdater的Update方法是rolling update的核心，它以上述config对象作为参数，其核心流程是每次让新RC的Pod副本数量加1，同时旧RC的Pod副本数量减1，直到新RC的Pod副本数量达到预期值同时旧RC的Pod副本数量变为零为止，在这个过程中由于新旧RC的Pod副本数量一直在变动，所以需要有一个地方记录最初不变的那个Pod副本数量，这里就是RC的Annotation标签——`kubectrl.kubernetes.io/desired-replicas`。

下面这段源码就是“贴标签”的过程：

```
    fmt.Fprintf(out, "Creating %s\n", newName)
        if newRc.ObjectMeta.Annotations == nil {
            newRc.ObjectMeta.Annotations =
map[string]string{}
        }

    newRc.ObjectMeta.Annotations[desiredReplicasAnnotation] =
fmt.Sprintf("%d", desired)

    newRc.ObjectMeta.Annotations[sourceIdAnnotation] = sourceId
        newRc.Spec.Replicas = 0
                                newRc, err =
r.c.CreateReplicationController(r.ns, n
```

下面这段源码便是“江山代有才人出，一代新人换旧人”的生动画面：

```
        for newRc.Spec.Replicas < desired &&
oldRc.Spec.Replicas != 0 {
            newRc.Spec.Replicas += 1
            oldRc.Spec.Replicas -= 1
            fmt.Printf("At beginning of loop: %s
replicas: %d, %s replicas: %d\n",
                        oldName, oldRc.Spec.Replicas,
                        newName, newRc.Spec.Replicas)
            fmt.Fprintf(out, "Updating %s replicas: %d,
%s replicas: %d\n",
                        oldName, oldRc.Spec.Replicas,
                        newName, newRc.Spec.Replicas)
            newRc, err = r.scaleAndWait(newRc, retry,
waitForReplicas)
            if err != nil {
                return err
            }
            time.Sleep(updatePeriod)
            oldRc, err = r.scaleAndWait(oldRc, retry,
waitForReplicas)
            if err != nil {
                return err
            }
            fmt.Printf("At end of loop: %s replicas:
```

```

%d, %s replicas: %d\n",
                                oldName, oldRc.Spec.Replicas,
                                newName, newRc.Spec.Replicas)
    }
    // delete remaining replicas on oldRc
    if oldRc.Spec.Replicas != 0 {
        fmt.Fprintf(out, "Stopping %s replicas: %d
-> %d\n",
                                oldName, oldRc.Spec.Replicas, 0)
        oldRc.Spec.Replicas = 0
        oldRc, err = r.scaleAndWait(oldRc, retry,
waitForReplicas)
        if err != nil {
            return err
        }
    }
    // add remaining replicas on newRc
    if newRc.Spec.Replicas != desired {
        fmt.Fprintf(out, "Scaling %s replicas: %d -
> %d\n",
                                newName, newRc.Spec.Replicas,
desired)
        newRc.Spec.Replicas = desired
        newRc, err = r.scaleAndWait(newRc, retry,
waitForReplicas)
        if err != nil {
            return err

```

```
    }  
}
```

上述方法里的 `scaleAndWait` 方法调用了 `kubectl.ReplicationControllerScaler` 的 `Scale` 方法，`Scale` 方法先通过 `Rest API` 调用 `Kubernetes API Server` 更新 `RC` 的 `Pod` 副本数量，然后循环拉取 `RC` 信息，直到超时或者 `RC` 同步状态完成。下面是判断 `RC` 同步状态是否完成的函数，来自 `client` 包（`pkg/client/conditions.go`）。

```
func ControllerHasDesiredReplicas(c Interface,  
controller *api.ReplicationController) wait.ConditionFunc {  
    desiredGeneration := controller.Generation  
    return func() (bool, error) {  
        ctrl, err :=  
c.ReplicationControllers(controller.Namespace).Get(controller.Name)  
        if err != nil {  
            return false, err  
        }  
        return ctrl.Status.ObservedGeneration >=  
desiredGeneration && ctrl.Status.Replicas ==  
ctrl.Spec.Replicas, nil  
    }  
}
```

`rolling-update` 是 `kubectl` 所有命令中最为复杂的一个，从它的功能和流程来看，完全可以被当作一个 `Job` 并放到 `kube-controller-manager` 上

实现，客户端仅仅发起Job的创建及Job状态查看等命令即可，未来Kubernetes的版本是否会这样重构，我们拭目以待。

后记

Kubernetes无疑是容器化技术时代最好的分布式系统架构，但是目前它还没有一款很好的图形化管理工具，基本上是命令行操作，因此不容易入门。另外，在系统运行过程中，我们难以直观了解当前服务的分布情况及资源的使用情况，日志也不完善，难以快速追踪和排查故障，因此，我们发起了一个名为**Ku8eye**的开源项目，这是借鉴了OpenStack Horizon、Cloudera Manager等知名软件的设计思想的一款国产开源软件，目标是成为Kubernetes的姊妹开源项目。

Ku8eye作为Kubernetes的一站式管理工具，具备如下关键特性。

- 图形化一键安装和部署多节点Kubernetes集群。这是安装、部署谷歌Kubernetes集群的最快、最佳方式，其安装流程会参考当前的系统环境，提供默认优化的集群安装参数，实现最佳部署。
- 支持多角色、多租户的Portal管理界面。通过一个集中化的Portal界面，运营团队可以很方便地调整集群配置及管理集群资源，实现跨部门的角色、用户及多租户管理，通过自助服务可以很容易完成Kubernetes集群的运维管理工作。
- 制定了Kubernetes应用的程序发布包标准（ku8package），并提供了一款向导工具，使得专门为Kubernetes设计的应用能够很容易地从本地环境发布到公有云和其他环境中；并且提供了

Kubernetes应用的可视化构建工具，实现了Kubernetes Service、RC、Pod及其他资源的可视化构建和管理功能。

- 可定制化的监控和告警系统。Ku8eye内建了很多系统健康检查工具来检测、发现异常并触发告警事件，不仅可以监控集群中的所有节点和组件（包括Docker与Kubernetes），还可以很容易地监控业务应用的性能；并且提供了一个强大的Dashboard，用来生成各种复杂的监控图表以展示历史信息，还可来自定义相关监控指标的告警阈值。
- 具备综合的全面的故障排查能力。Ku8eye提供了集中化的唯一日志管理工具，日志系统从集群中的各个节点拉取日志并做聚合分析，拉取的日志包括系统日志和用户程序日志；并且提供了全文检索能力以方便故障分析和问题排查，检索的信息包括相关告警信息，而历史视图和相关的度量数据则告诉我们什么时候发生了什么事情，有助于快速了解相关时间内系统的行为特征。
- 实现了Docker与Kubernetes项目的持续集成功能。Ku8eye提供了一款可视化工具，用来驱动持续集成的整个流程，包括创建新的Docker镜像，Push镜像到私有仓库，创建Kubernetes测试环境进行测试，以及最终滚动升级到生产环境中的各个主要环节。

Ku8eye的GitHub地址为<https://github.com/bestcloud>，Ku8eye目前所用到的技术包括Java Web、Ansible脚本，未来可能涉及Python脚本及Android开发等。截至本书出版时，Ku8eye已有10多名团队成员。如果您有兴趣，可在学完本书后加入本项目QQ群（Kubernetes中国）：285431657。

People

12 >



Invite someone

请您加入我们